NAME

libthread_db, td_log, td_ta_new, td_ta_delete, td_init, td_ta_get_ph, td_ta_get_nthreads, td_ta_tsd_iter, td_ta_thr_iter, td_thr_validate, td_thr_tsd, td_thr_get_info, td_thr_getfpregs, td_thr_getxregsize, td_thr_getxregs, td_thr_sigsetmask, td_thr_setprio, td_thr_setsigpending, td_thr_setfpregs, td_thr_setxregs, td_ta_map_id2thr, td_ta_map_lwp2thr, td_thr_getgregs, td_thr_setgregs – interface to libthread threads information

SYNOPSIS

cc [*flag* ...] *file* ... **/lib/libthread_db.so.1** [*library* ...]

#include <proc_service.h>
#include <thread_db.h>

void td_log(const int *on_off***);**

td_err_e td_ta_new(const struct ps_prochandle **ph_p***, td_thragent_t** ***ta_pp***);**

td_err_e td_ta_delete(const td_thragent_t **ta_p***);**

td_err_e td_init();

td_err_e td_ta_get_ph(const td_thragent_t **ta_p***, struct ps_prochandle** ***ph_pp***);**

td_err_e td_ta_get_nthreads(const td_thragent_t **ta_p***, int** **nthread_p***);**

td_err_e td_ta_tsd_iter(const td_thragent_t **ta_p***, td_key_iter_f** **cb***, void** **cbdata_p***);**

td_err_e td_ta_thr_iter(const td_thragent_t **ta_p***, td_thr_iter_f** **cb***, void** **cbdata_p***,**
td_thr_state_e *state***, int** *ti_pri,* **sigset_t** **ti_sigmask_p***, unsigned** *ti_user_flags***);**

td_err_e td_thr_validate(const td_thrhandle_t **th_p***);**

td_err_e td_thr_tsd(const td_thrhandle_t **th_p***, const thread_key_t** *key***,**
void ***data_pp***);**

td_err_e td_thr_get_info(const td_thrhandle_t **th_p***, td_thrinfo_t** **ti_p***);**

td_err_e td_thr_getfpregs(const td_thrhandle_t **th_p***, prfpregset_t** **fpregset***);**

td_err_e td_thr_getxregsize(const td_thrhandle_t **th_p***, int** **xregsize***);**

td_err_e td_thr_getxregs(const td_thrhandle_t **th_p***, const caddr_t** **xregset***);**

td_err_e td_thr_sigsetmask(const td_thrhandle_t **th_p***, const sigset_t** *ti_sigmask***);**

td_err_e td_thr_setprio(const td_thrhandle_t **th_p***, const int** *ti_pri***);**

td_err_e td_thr_setsigpending(const td_thrhandle_t **th_p***,**
const uchar_t *ti_pending_flag***, const sigset_t** *ti_pending***);**

td_err_e td_thr_setfpregs(const td_thrhandle_t **th_p***, const prfpregset_t** **fpregset***);**

td_err_e td_thr_setxregs(const td_thrhandle_t **th_p***, const caddr_t** **xregset***);**

td_err_e td_ta_map_id2thr(const td_thragent_t **ta_p***, thread_t** *tid***,**
td_thrhandle_t **th_p***);**

td_err_e td_ta_map_lwp2thr(const td_thragent_t **ta_p***, lwpid_t** *lwpid***,**
td_thrhandle_t **th_p***);**

td_err_e td_thr_getgregs(const td_thrhandle_t **th_p*, **prgregset_t** *regset***);**

td_err_e td_thr_setgregs(const td_thrhandle_t **th_p*, **const prgregset_t** *regset***);**

DESCRIPTION

libthread_db is a library of functions for accessing information about threads in a process under inspection (PUI). The typical use for this library is by a debugger that requires information about the threads in a process. To some extent it can also be used for self inspection of threads within the same process. libthread_db utilizes functions provided by the user for interrogating a process (i.e., reading, writing, stopping, continuing, etc. a process). The prototypes for these user provided functions, referred to as the proc_service functions are described in <proc_service.h>. libthread_db should be accessed by an explicit version of the library (e.g., libthread_db.so.1) to ensure the correct version is used.

Support Level

All interfaces in libthread_db are UNCOMMITTED interfaces and are subject to change in future releases.

FUNCTIONS

td_log( ) turns logging on and off. A function in the proc_service interface is provided by the user for logging execution points in libthread_db. If logging is turned on, this proc_service function is called. The nominal name of the function is **ps_plog**. *on_off* = 0 turns logging off, non-zero turns logging on. proc_service functions called: none.

td_ta_new( ) allocates a thread agent for the given process handle and return a pointer to it. proc_service functions called: ps_pglobal_lookup.

td_ta_delete( ) deallocates the thread agent. proc_service functions called: none.

td_init( ) performs initialization for libthread_db interface. proc_service functions called: none

td_ta_get_ph( ) gets the process handle out of a thread agent and returns it. proc_service functions called: none

td_ta_get_nthreads( ) gets the total number of threads in a process. This number includes both user and system threads. proc_service functions called: **ps_pglobal_lookup**, **ps_pdread**, **ps_pstop**, **ps_pcontinue**.

td_ta_tsd_iter( ) iterates over the set of global TSD keys. The call back function is called with three arguments, a key, a pointer to a destructor function, and an extra pointer which can be NULL depending on the call back. The call back function is called once for each key. If return value of *cb* is non-zero, terminate iterations. proc_service functions called: *ps_pglobal_lookup*, *ps_pdread*, *ps_pstop*, *ps_pcontinue*.

td_ta_thr_iter( ) iterates over all threads. For each thread call the function pointed to by *cb* with a pointer to a thread handle, and a pointer to data which can be NULL. Only call *cb on threads* which match the properties of *state*, *ti_pri*, *ti_sigmask_p*, and *ti_user_flags*. If *cb* returns a non-zero value, iterations terminate. The call back function is defined by user. **td_thr_iter_f** takes a thread handle and *cbdata_p* as a parameter. *state* is the state of threads of interest. A value of **TD_THR_ANY_STATE** from enum **td_thr_state_e** does not restrict iterations by state. *ti_pri* is the lower bound of priorities of threads of interest. A value of **TD_THR_LOWEST_PRIORITY** defined in thread_db.h does not restrict iterations by priority. A thread with priority less than *ti_pri* will NOT be passed to the callback function. *ti_sigmask_p* is the signal mask of threads of interest. A value of **TD_SIGNO_MASK** defined in thread_db.h does not restrict iterations by signal mask. *ti_user_flags* is the user flags of threads of interest. A value of **TD_THR_ANY_USER_FLAGS** defined in thread_db.h does not restrict iterations by user flags. proc_service functions called: *ps_pdread*, *ps_pstop*, *ps_pcontinue*.

td_thr_validate( ) validates the thread handle. **td_thr_validate( )** checks that a thread exists in the thread agent/process that corresponds to thread with handle **th_p*. Return value: **TD_OK** implies thread handle is valid. **TD_NOTHR** implies thread handle not valid. proc_service functions called: *ps_pdread*, *ps_pstop*, *ps_pcontinue*.

td_thr_tsd( ) gets a thread's private binding to a given thread specific data (TSD) key (see **thr_getspecific**(3T)). If the thread doesn't have a binding for a particular key, then NULL is returned. proc_service functions called: *ps_pglobal_lookup*, *ps_pdread*, *ps_pstop*, *ps_pcontinue.*

td_thr_get_info( ) updates the thread information struct. All fields in a thread information structure (**td_thrinfo_t**) will be updated to be consistent with properties of its respective thread. proc_service functions called: **ps_pdread**, **ps_pstop**, **ps_pcontinue**.

td_thr_getfpregs( ) gets the floating point registers for the given thread. proc_service functions called: **ps_pdread**, **ps_pstop**, **ps_pcontinue**, **ps_lgetfpregs**.

td_thr_getxregsize( ) gets the size of the extra register set for the given thread. proc_service functions called: **ps_pdread**, **ps_pstop**, **ps_pcontinue**, **ps_lgetxregsize**.

td_thr_getxregs( ) gets the extra registers for the given thread. proc_service functions called: **ps_pdread**, **ps_pstop**, **ps_pcontinue**, **ps_lgetxregs**.

td_thr_sigsetmask( ) changes a thread's signal mask to the value specified by *ti_sigmask*. proc_service functions called: **ps_pstop**, **ps_pcontinue**, **ps_pdwrite**.

td_thr_setprio( ) changes a thread's priority to the value specified by **ti_pri.** proc_service functions called: *ps_pdwrite*.

td_thr_setsigpending( ) changes a thread's pending signal state to that specified by *ti_pending_flag* and *ti_pending.* A null value for *ti_pending_flag* indicates that there are no pending signals for the thread. proc_service functions called: **ps_pstop**, **ps_pcontinue**, **ps_pdwrite**.

td_thr_setfpregs( ) sets the floating pointing registers for a given thread. proc_service functions called: **ps_pdread**, **ps_pstop**, **ps_lsetfpregs**, **ps_pcontinue**.

td_thr_setxregs( ) sets the extra registers for the given thread. proc_service functions called: **ps_pcontinue**, **ps_pstop**, **ps_pdread**, **ps_pdwrite**, **ps_lsetxregs**.

td_ta_map_id2thr( ) returns the thread handle corresponding to the given thread identifier tid. proc_service functions called: **ps_pdread**, **ps_pstop**, **ps_pcontinue**.

td_ta_map_lwp2thr( ) returns a thread handle for the given thread agent and lwp id. proc_service functions called: **ps_pdread**, **ps_lgetgregs**.

td_thr_getgregs( ) gets the general registers for a given thread. For a thread that is currently executing on an LWP, (**td_thr_state_e**) **TD_THR_ACTIVE**, all registers in *regset* will be read for the thread. For a thread not executing on an LWP, only the following registers will be read.

SPARC

%i0-%i7,
%l0-%l7,
%g7, %pc, %sp (%o6).

x86

%ebp, %edi, %esi, %ebx, %sp, %pc.

%pc and %sp will be the program counter and stack pointer at the point where the thread will resume execution when it becomes active, (**td_thr_state_e**) **TD_THR_ACTIVE**. proc_service functions called: **ps_pstop**, **ps_pcontinue**, **ps_pdread**, **ps_lgetregs**.

td_thr_setgregs( ) sets the general registers for a given thread. For a thread that is currently executing on an LWP, (**td_thr_state_e**) **TD_THR_ACTIVE**, all registers in regset will be written for the thread. For a thread not executing on an LWP, only the following registers will be written.

SPARC

%i0-%i7,
%l0-%l7,
%pc, %sp (%o6).

x86

%ebp, %edi, %esi, %ebx, %sp, %pc.

%pc and %sp will be the program counter and stack pointer at the point where the thread will resume execution when it becomes active, (**td_thr_state_e**) **TD_THR_ACTIVE**. proc_service functions called: **ps_pstop**, **ps_pcontinue**, **ps_pdread**, **ps_lsetregs**.

RETURN VALUES

TD_OK The call completed successfully.

TD_ERR The call failed and no other error code applies.

TD_NOTHR The call failed because the thread handle passed as a parameter does not correspond to any thread known to libthread_db. The thread may no longer exist or the user may be using a corrupted thread handle.

TD_NOSV If the synchronization variable passed as a parameter does not correspond to any synchronization variable known to libthread_db the call will return this value. The synchronization variable may no longer exist or the user may be using a corrupted synchronization handle.

TD_NOLWP If there is no LWP that is part of the process that corresponds to the given lwpid the call will fail and return this value. This can occur if the lwpid is corrupted or if the lwp has been returned to the OS.

TD_BADPH The call failed because the process handle is invalid. This generally indicates that a NULL values was passed as a process handle.

TD_BADTH The call failed because the thread handle is invalid.

TD_BADSH The call failed because the synchronization handle is invalid.

TD_BADKEY The call failed because the key for the thread specific data area is invalid.

TD_NOMSG If there is not event message information, this value is returned.

TD_NOFPREGS If the floating point registers are not available for this thread the call returns this value.

TD_NOLIBTHREAD
If the applications is not threaded (i.e., not linked with libthread) this value is returned.

TD_NOEVENT This value is returned if an address was requested for an event which is not supported.

TD_NOCAPAB This value indicates that the requested service is not supported. This generally indicates a limitation with the current library that will be rectified in a future release.

TD_DBERR The call failed because an error occurred during a request for service from the user's proc_service interface.

TD_NOAPPLIC This value is returned if the requested function does not apply to the given variable.

TD_NOTSD The call failed because there is no thread specific data for this thread. A key may exist without an entry for every thread.

TD_MALLOC The call failed because a malloc failed.

TD_PARTIALREG
An operation on a register set was performed on only part of the register set. For example, at some points in the execution of a process not all the general purpose registers may be available.

TD_NOXREGS If the extra registers are not available for this thread the call returns this value.

EXAMPLES

```
/*
*  Specify libthread_db.so.1 on link line to access correct version.
*  cc thisfile.c /lib/libthread_db.so.1
*/
#include <stdio.h>
#include <sys/types.h>
#include <thread_db.h>
#include <dlfcn.h>

static int thread_cb( const td_thrhandle_t *th_p, void *s );
struct ps_prochandle {
  pid_t pid;
};
```

```
struct ps_prochandle ph = {1};
td_thragent_t *ta_p;

/*
*   libthread_db example.
*      Initialize libthread_db
*      Create a thread agent
*      Call thread iterator
*/
int main()
{
  td_err_e td_return;
  /*
  *   td_init()
  */
  td_return = td_init();
  if ( td_return != TD_OK ) {
    printf("Initialization error on td_init() call\n");
    return 0;
  }
  /*
  *   td_ta_new()
  */
  td_return = td_ta_new(&ph, &ta_p);
  if ( td_return == TD_OK ) {
    /*
    * td_ta_thr_iter()
    */
    (void) td_ta_thr_iter(ta_p, thread_cb, "Import calls test",
      TD_THR_ANY_STATE, TD_THR_LOWEST_PRIORITY,
      TD_SIGNO_MASK, TD_THR_ANY_USER_FLAGS);
  }

  return 0;
}

static int thread_cb( const td_thrhandle_t *th_p, void *s )
/*
*  Description:
*     Call back function for iterator
*
*  Input:
*     *th_p - thread handle
*     *s - data
*
```

```
*  Output:
*     none
*
*/
{
   int return_val = 0;
   td_err_e td_return;
   td_thrinfo_t to;
   td_return = td_thr_get_info(th_p, &(to));
   if ( td_return == TD_ERR ) {
      printf("Thread update failed\n");
      return_val = 1;
   }
   return return_val;
}

/*
*   Use dlopen and dlsym to access libthread_db functions.
*/
#include <stdio.h>
#include <sys/types.h>
#include <thread_db.h>
#include <dlfcn.h>

static int thread_cb( const td_thrhandle_t *th_p, void *s );
struct ps_prochandle {
   pid_t pid;
};
struct ps_prochandle ph = {1};
td_thragent_t *ta_p;

td_err_e (*init)(void);
td_err_e (*ta_new)(const struct ps_prochandle *ph_p, td_thragent_t **ta_pp);
td_err_e (*thr_iter)(const td_thragent_t *ta_p, td_thr_iter_f *cb,
              void *cbdata_p, td_thr_state_e state, int ti_pri,
              sigset_t *ti_sigmask_p, unsigned ti_user_flags);
td_err_e (*get_info)(const td_thrhandle_t *th_p, td_thrinfo_t *ti_p);

/*
*   libthread_db example.
*      Initialize libthread_db
*      Create a thread agent
*      Call thread iterator
*/
int main()
```

```
{
  td_err_e td_return;
  void *handle;
  /*
  *   Access a specific version of libthread_db.
  */
  handle = dlopen("libthread_db.so.1", RTLD_LAZY);
  if ( !handle ) {
    printf("dlopen error: %s\n", dlerror());
    return 1;
  }
  else {
    init = (td_err_e (*)(void))dlsym(handle, "td_init");
    ta_new = (td_err_e (*)(const struct ps_prochandle *, td_thragent_t **))
          dlsym(handle, "td_ta_new");
    thr_iter = (td_err_e (*)(const td_thragent_t *, td_thr_iter_f *,
          void *, td_thr_state_e , int , sigset_t *, unsigned ))
          dlsym(handle, "td_ta_thr_iter");
    get_info = (td_err_e (*)(const td_thrhandle_t *, td_thrinfo_t *))
          dlsym(handle, "td_thr_get_info");
  }

  /*
  *   td_init()
  */
  td_return = (*init)();
  if ( td_return != TD_OK ) {
    printf("Initialization error on td_init() call\n");
    return 0;
  }
  /*
  *   td_ta_new()
  */
  td_return = (*ta_new)(&ph, &ta_p);
  if ( td_return == TD_OK ) {
    /*
    * td_ta_thr_iter()
    */
    (void) (*thr_iter)(ta_p, thread_cb, "Import calls test",
      TD_THR_ANY_STATE, TD_THR_LOWEST_PRIORITY,
      TD_SIGNO_MASK, TD_THR_ANY_USER_FLAGS);
  }

  return 0;
}
```

```
static int thread_cb( const td_thrhandle_t *th_p, void *s )
/*
* Description:
*    Call back function for iterator
*
* Input:
*    *th_p - thread handle
*    *s - data
*
* Output:
*    none
*
*/
{
  int return_val = 0;
  td_err_e td_return;
  td_thrinfo_t to;
  td_return = (*get_info)(th_p, &(to));
  if ( td_return == TD_ERR ) {
    printf("Thread update failed\n");
    return_val = 1;
  }
  return return_val;
}
```

FILES **/usr/lib/libthread_db.so.1**

SEE ALSO **proc_service**(3T), **thr_getspecific**(3T)