Twizzler Documentation

Documentation version 0.2-1; Twizzler version 0.2

Contents

Twizzler Overview

The Twizzler Operating System is designed for non-volatile memory and to unify the hardware programming model with the software programming model. This means that it is, fundamentally, a single-level store, exokernel-like system that:

- Places most of the system in userspace.
- Has *very* limited in-kernel blocking.
- Allows hardware to act autonomously (if capable).
- Provides persistent object support as the primary data abstraction.

Twizzler does away with a number of traditional abstractions that a Unix has (note we provide enough Unix emulation to have a useable C runtime and standard library, and many POSIX functions do work):

- Files are not a thing in Twizzler. Instead, Twizzler is based around *objects*. Threads can access objects through load and store instructions (they are memory objects). Persistence is automatic and configurable.
- There is no notion of a process. Instead, Twizzler provides the core components of a process as separate, individually controllable abstractions. These include *views* (an abstraction for virtual-memory machines), *security contexts* (which control access for hardware and software to objects, and restrict control-flow transfers), and *threads* (which are similar to threads in standard OSes).
- File systems are not needed on Twizzler. Objects are accessed by an ID (no file handles, no inodes). When names are needed, name resolvers can be used, but the naming system is disconnected from data storage.

- Traditional Unix users, groups, etc., are not used in Twizzler. Instead, we provide a more robust and fine-grained security mechanism based on cryptographically signed capabilities, security contexts, and content-derived names.

Twizzler programs must be *statically linked* at the moment. This limitation will soon be removed. A Twizzler program typically has the following core components:

- The program itself, the executable.
- The program is linked to musl, a C library, to provide a C standard library.
- The program is also linked to Twix (libtwix), a library that emulates the Linux system call interface (needed for musl to be happy).
- Finally, we link to libtwz, the Twizzler standard library, which provides a runtime for Twizzler (eg. fault handling, userspace-level object management, etc).

Twizzler's primary abstractions, of which there are few, build the base of the system. Further structure can be imposed on objects in userspace, but the core system provides only the following:

- Basic object manipulation. Objects are flat memory spaces of data, with a meta-data page at the end. The kernel understands some of the meta-data page, as it is used for ID derivation and security purposes. Objects can be created, deleted, and configured. See doc/us/objects.md.
- Pointer manipulation. Loading, resolving, storing, etc. See doc/us/pointers.md.
- Kernel State Objects (KSOs). A KSO is an object whose format has a specified internal layout that the kernel understands. These are used to configure the running kernel, or reload a previous running state. These include:
  - Views
  - Security Contexts
  - Root KSOs
  - Threads
  - Devices

  See doc/core/kso.md for details.
- Thread control. Threads can be created, waited on, can exit, can signal other threads, and can synchronize. See doc/us/threads.md.
- Structured Objects. Twizzler provides a mechanism for generically adding structure that can be discovered inside an object. See us/structured_objects.md. The core structured objects Twizzler provides are:
  - bstream (a simplex, byte-oriented stream).
  - dgram (a duplex data-gram interface).
  - event (a system for waiting on events, similar to select/poll/epoll).
  - bstreamd (a duplex, byte-oriented stream).
  - user (a user file, describing a user).
  - kring (a key-ring object).
  - ku (a public key object).
  - kr (a private key object).

  See the associated doc/us/*type*.md file for details on these.

Object IDs (doc/us/object_ids.md)

All objects in the system have a 128 bit GUID (or ID). These IDs are globally unique, and are randomly generated. The header `twz/obj.h` provides the type `objid_t` which can store object IDs.

Typically it is preferred for applications to use object handles (see Object API) instead of IDs directly.

Printing object IDs

Twizzler provides an object ID format specifier for printf-like functions: `IDFMT` and `IDPR(objid_t id)`. They are used as follows:

```
objid_t id = /* something */;
printf("Here is the ID: " IDFMT "\n", IDPR(id));
```

Object API (us/objects.md)

Twizzler objects have a maximum size (found in `OBJ_MAXSIZE`). Objects are broken into two regions: *data* and *metadata*. This distinction is largely irrelevant, except that `libtwz` often manipulates metadata, and metadata grows downward from the top of the object to the end of the data region (`OBJ_TOPDATA`). The data region starts at 0 and grows upward. Near the top of the object (in a known location) is a structure of type `struct metainfo`. This struct contains metadata information. Twizzler provides functions to manipulate its contents.

Note that the first page of an object is unmapped (for catching NULL pointers). Twizzler exposes a constant, `OBJ_NULLPAGE_SIZE`, that indicates how large this is. Normal programs will not need to use this constant most of the time. All offsets within an object must be larger than this constant.

Objects are referred to by programs through object handles of type twzobj. Typically, Twizzler API functions will take a pointer to a twzobj handle as an input. You can think of these as similar to FILE in C, though with much less state.

Most Twizzler functions return an error code directly, instead of the standard C approach of using errno. The error codes are typically negative numbers from the errno header. Note that C library functions will still use errno. Some functions are designed to "succeed or throw", in which case they often return data directly or return void and will silently succeed or else throw an exception (see Exceptions).

twz_object_init_guid

```
#include <twz/obj.h>
int twz_object_init_guid(twzobj *obj, objid_t id, int flags);
```

Initialize an object handle that refers to the object specified by `id`. The `flags` argument is a bitfield with the following bits:

- `FE_READ`: Read access requested
- `FE_WRITE`: Write access requested
- `FE_EXEC`: Execute access requested
- `FE_DERIVE`: Open a copy of this object, not the object itself.

If the calling thread does not have the permissions requested, the function might still succeed. First access to the data that is disallowed will cause a fault.

Return Value

Returns 0 on success, and error code on error.

Errors

- `-EINVAL`: Invalid argument.
- `-ENOENT`: Could not locate object `id`. Note that if `id` does not exist, this function is not required to return an error.
- `-EACCES`: Tried to request both write and execute permissions simultaneously.
- `-ENOMEM`: Not enough memory to fulfill the request.

twz_object_init_name

```
#include <twz/obj.h>
int twz_object_init_name(twzobj *obj, const char *name, int flags);
```

Initialize an object handle that refers to the object specified by name `name` after being passed to the default name resolver. The flags are the same as `twz_object_init_guid`.

Return Value

Returns 0 on success, and error code on error.

Errors

In addition to the errors returned by `twz_object_init_guid`, this function can return:

- `-ENOENT`: The name was not resolved successfully.
- `-ELOOP`: The name resolution was recursive and too deep.

twz_object_new

```
#include <twz/obj.h>
int twz_object_new(twzobj *newobj, twzobj *srcobj, twzobj *kuid, int flags);
```

Create an object, using newobj as the handle to the new object. The other arguments are as follows:

- `srcobj`: Copy from a *source object*; that is, the new object will be byte-wise identical to the source object specified by this argument (with the exception of some metadata fields). If NULL, the new object is initialized to contain zeros, except key metadata info.
- `kuid`: Give the object a public-key that can be used to verify signatures for capabilities for this object (see Security). If NULL, the new object will not have a public-key. If special value `TWZ_KU_USER`, the function will assign the new object's public key as the contents of the env var `TWZUSERKU`, thus providing a default.
- `flags` is a bitwise combination of the following:
  - `TWZ_OC_HASHDATA`: When deriving the object ID, hash the object's data. This makes the object immutable. It cannot be combined with the `TWZ_OC_DFL_WRITE` flag.
  - `TWZ_OC_DFL_*`: For values of `*` being `READ`, `WRITE`, `EXEC`, `DEL`, and `USE`, this sets the default permissions of the object. For example, anyone who knows the ID of an object with `TWZ_OC_DFL_WRITE` set can write to the object.
  - `TWZ_OC_VOLATILE`: make this object volatile. A *persistent* object is guaranteed to be reachable across reboots. A volatile object is *not* guaranteed this. TODO: is to worth strengthening this?
  - `TWZ_OC_TIED_NONE`: Do not tie this object's lifetime to another object. By default, a new object's lifetime is tied to the creating thread. For more details, see Tying Objects.
  - `TWZ_OC_TIED_VIEW`: Tie the object to the current view ("wire" the object). This flag is incompatible with `TWZ_OC_TIED_NONE`.

Errors

In addition to errors returned by `sys_ocreate` and `twz_object_init_guid`, this function can return:

- `-EINVAL`: The user specified `TWZ_KU_USER` for `kuid` and Twizzler failed to determine the correct GUID for the public-key.

twz_object_release

```
#include <twz/obj.h>
int twz_object_release(twzobj *obj);
```

Destroy this object handle. This call is idempotent for a given handle. This call releases userspace-level resources for the given object. It may include a system-call, but is not guaranteed to.

Return Value

Returns 0 on success, error code on error.

Errors

- `-EINVAL`: Invalid argument.

twz_object_base

```
#include <twz/obj.h>
void *twz_object_base(twzobj *obj);
```

Return a d-ptr (see Pointer Manipulation; essentially, a pointer which can be dereferenced) that refers to the base of the object (the first data byte).

Return Value

On success, return a pointer to the object's data start. Throws `FAULT_PPTR` on error.

twz_object_meta

```
#include <twz/obj.h>
struct metainfo *twz_object_meta(twzobj *obj);
```

Similar to `twz_object_base`, but return a pointer to the metainfo structure instead.

twz_object_delete

```
#include <twz/obj.h>
int twz_object_delete(twzobj *obj, int flags);
```

Delete an object. The object will be deleted by the kernel when all internal references are dropped. Note that this DOES NOT mean pointers to data within this object. An internal kernel reference just refers to mappings in the address spaces, etc. If you want to delete an object and keep it around for some time, see Tying Objects (this lets you do the unlink-after-open trick for automatic cleanup from Unix). By default, after an object has been deleted by this call but not yet cleaned up (due to internal references), new references to this object can still be created (it can be opened and used normally).

Flags is a bitwise combination of the following:

- `TWZ_OD_IMMEDIATE`: New references cannot be created to this object.

Return Value

Returns 0 on success, error code on error.

Errors

See `sys_odelete`.

Object Metadata

The object's metadata contains three primary compenents:

- The metainfo struct, located at the base of the last page of the object.

- The foreign object table, which contains a list of object IDs used by cross-object pointers to refer outside of the object.
- A list of structured object tags (see Structured Objects).

Pointer Manipulation (us/pointers.md)

Pointers have two forms, referred to as *persistent* and *dereferencable*. A persistent pointer (p-ptr) is a pointer of the form `(fot-entry, offset)`. A dereferencable pointer (d-ptr) is a pointer in a form that the CPU can directly use to access data behind it. In terms of C, a d-ptr can have a * applied to it, whereas a p-ptr must first be converted into a d-ptr.

Note that some internal code may still refer to d-ptr as v-ptr.

Pointer manipulation routines take d-ptr's by default for pointer arguments.

twz_object_lea

```
#include <twz/obj.h>
T *twz_object_lea(twzobj *obj, T *ptr [p-ptr]);
```

Computes a d-ptr from a p-ptr. The `obj` argument must not be NULL, and refers to the object whose FOT will be used for the computation. The method of computation is platform-specific, and could be a no-op. If `ptr` is NULL, no computation is performed, and NULL is returned.

Return Value

On successful computation, a d-ptr that accesses the data specified by the p-ptr is returned. The d-ptr is valid for the same lifetime as `obj`.

Errors

This function does not return errors, and will either succeed or raise a fault. The possible faults can be:

- `FAULT_PPTR`: An error occurred attempting to resolve the specified FOT entry in the pointer. Possible info codes:
  - `FAULT_PPTR_RESOURCES`: The entry exceeded the number of FOT entries in this object.
  - `FAULT_PPTR_RESOURCES`: Resources exhausted for d-ptrs.
  - `FAULT_PPTR_INVALID`: The FOT entry specified is invalid.
  - `FAULT_PPTR_RESOLVE`: The FOT entry had a name which failed to resolve.

twz_ptr_local

```
#include <twz/obj.h>
T *twz_ptr_local(T *ptr [p-ptr]);
```

Computes the offset into the object that the ptr refers to, essentially converts the pointer into a local pointer.

Return Value

Returns a local p-ptr (fot-entry = 0) for the specified pointer.

Errors

None.

twz_ptr_store_guid

```
#include <twz/obj.h>
int twz_ptr_store_guid(twzobj *obj, const T **loc, twzobj *target,
                       const T *p, uint64_t flags);
```

Create a p-ptr from `p`, store it in the location pointed to by `loc` (which must be a d-ptr, and must refer to a location within the object specified by `obj`). If `target` is NULL, `p` is interpreted as a d-ptr, otherwise `p` is interpreted as a p-ptr within object `target`. The `flags` argument specifies the flags to store in the FOT entry for this pointer (see twz_object_init_guid).

Return Value

Returns 0 on success, error code on error.

Errors

- `-ENOENT`: `p` was a d-ptr, but could not be matched to an existing object.
- `-EINVAL`: Invalid argument.
- `-ENOSPC`: No more FOT entries are available in `obj`.

twz_ptr_store_name

```
#include <twz/obj.h>
int twz_ptr_store_name(twzobj *obj, const T **loc, const char *name,
                       const T *p, uint64_t flags);
```

WARNING - this API is unstable.

Create a p-ptr from `p`, store it in the location pointed to by `loc` (which must be a d-ptr, and must refer to a location within the object specified by `obj`). The FOT entry for this p-ptr will refer to an object by name `name`. `p` must be a p-ptr. See `twz_ptr_store_guid` for flags.

Return Value

Returns 0 on success, error code on error.

Errors

- `-EINVAL`: Invalid argument.
- `-ENOSPC`: No more FOT entries are available in `obj`.

twz_ptr_swizzle

```
#include <twz/obj.h>
[p-ptr] T *twz_ptr_swizzle(twzobj *obj, T *p, uint64_t flags);
```

Create and return a p-ptr that can be stored into object `obj`. This manipulates the FOT. `p` must be a d-ptr. See `twz_ptr_store_guid` for flags.

Return Value

Returns a p-ptr from that refers to the data pointed to by `p`. Throws on error.

Errors

- `FAULT_PPTR`: An error occurred attempting to resolve the specified FOT entry in the pointer. Possible info codes:
  - `FAULT_PPTR_RESOURCES`: Out of FOT entries.
  - `FAULT_PPTR_INVALID`: `p` was not a valid d-ptr.

Data Structure Consistency (us/consistency.md)

Twizzler provides a basic set of consistency primitives, under the hope that future language support will provide better consistency operations (much like what happened with atomics and threading!).

Detailed documentation on this is pending, but the basics are…

See us/include/twz/persist.h for the following functions:

- `_clwb(void *)`: Cache-line writeback a cache-line.
- `_clwb_len(void *, size_t)` Writeback a region of memory.
- `_pfence()`: Issue a fence to ensure consistency.

Transaction Logging

See us/include/twz/tx.h for a basic transactions system implemented as an undo log. NOTE: these transactions are not for coherence! They only work in single-threaded code, so if you want to use these in a multithreaded environment, put a mutex around it.

Your object needs to have a `struct twz_tx` object. This can be initialized with `twz_init(struct twz_tx *tx, size_t loglen);`

A transaction looks like this:

```
struct header *hdr = /*something*/;
TXSTART(obj, &hdr->tx) {
    TXRECORD(&hdr->tx, hdr->foo);
    hdr->foo = 3;
    if(want_to_commit)
        TXCOMMIT;
    else
        TXABORT(err_code);
} TX_ONABORT {
    /* do something with errno */
} TX_ONSUCCESS {
    /* we did it */
} TXEND;
```

The `TX_ONABORT` and `TX_ONSUCCESS` bits are optional.

Whenever you access transaction-protected variables outside of a transaction, you must first ensure that there is no pending transaction abort. You can do this with `TXCHECK(obj, tx)`.

Twizzler Thread API

WARNING - this API is unstable. If you really want threads, consider pthreads, fork, etc.

…threads have thread control objects… TODO

twz_thread_spawn

```
int twz_thread_spawn(struct thread *thrd, struct thrd_spawn_args *args);
```

Spawn a new thread, initializing the handle `thrd` for future use interacting with the thread. The new thread will be initialized according to `args`:

```
struct thrd_spawn_args {
    objid_t target_view;
    void (*start_func)(void *); /* thread entry function. */
    void *arg;                  /* argument for entry function. */
    char *stack_base;           /* stack base address. */
    size_t stack_size;
    char *tls_base; /* tls base address. */
};
```

If `target_view` is zero, the thread will be given the same view as the current thread. The thread will be started in `start_func` with argument `arg`.

Optionally, the thread can be given a stack base address and a stack size. If `stack_base` is set to NULL then the thread will be given a new stack object automatically.

Return Value

Errors

This function can return any error returned by `sys_thrd_spawn` or `twz_object_create`.

twz_thread_wait

```
ssize_t twz_thread_wait(struct thrd_wait_args *args, size_t count);
```

Wait for thread(s) on (a) particular sync point(s). All thread control objects have a collection of sync points. A sync point with value 0 indicates "not ready", whereas a sync point with a non-zero value indicates ready (and the value can be interpreted as some kind of info).

The function takes an array of `struct thrd_wait_args`, each of which defines a sync point to wait on for a thread. The function waits for any one of the specified sync points to be ready before returning. If a sync point is ready, the `info` field of the associated `struct thrd_wait_args` is filled out with the value of the sync point. The `event` field is set to a non-zero value as well.

```
struct thrd_wait_args {
    struct thread *thread;
    int syncpoint;
    long event;
    uint64_t info;
};
```

The `struct thrd_wait_args`'s field `thread` specifies which thread to wait for, and the `syncpoint` field specifies which sync point to wait for. Twizzler defines the following standard sync points:

- `THRD_SYNC_SPAWNED`: Set when the thread is spawned. Value is unspecified.
- `THRD_SYNC_READY`: Set when the thread is "ready". Value is unspecified. What ready means depends on the thread and the application. Often set when the thread is done with initialization.
- `THRD_SYNC_EXIT`: Set when the thread exits. The value is the exit code.

Return Value

Returns a count of how many sync points are ready on success, error code on error.

Errors

This function can return any error returned by `sys_thrd_sync`.

twz_thread_ready

```
int twz_thread_ready(struct thread *thread, int sp, uint64_t info);
```

Mark a sync point `sp` for thread `thread` as ready, using `info` as the value for the sync point. The `info` argument must be non-zero for threads to determine that `sp` is ready. If `thread` is NULL, use the current thread. Must have write access to the specified thread control object.

Return Value

Returns 0 on success, error code on error.

Errors

- `-EINVAL`: Invalid argument

twz_thread_repr_base

```
void *twz_thread_repr_base(void);
```

Determine a d-ptr to the base of the current thread's control object.

Return Value

Returns a pointer to the base of the current thread control object. This function always succeeds.

twz_stdstack

The `twz/thread.h` header provides an object, `twz_stdstack`, which is a `twzobj *` that refers to the standard stack object.

Naming in Twizzler (us/names.md)

"But Taborlin knew the names of all things, and so all things were his to command." - The Name of the Wind

WARNING - this API is unstable.

Naming in Twizzler is fundamentally a flexible concept. The idea of a name is not actually known by the *kernel*, nor most of the userspace APIs. They usually operate on pointers and IDs. Names must be first resolved into an ID to be *really* useful. The primary location names show up are in FOT entries, where the system resolves the name (according to the supplied name resolver) into an ID.

A name resolver is a function which takes a null-terminated string of bytes and returns an ID (or an error). A name is an unspecified C string.

twz_name_resolve

```
#include <twz/name.h>
int twz_name_resolve(twzobj *obj,
  const char *name,
  int (*fn)(twzobj *, const char *, int, objid_t *),
```

```
  int flags,
  objid_t *id);
```

Resolve a name into an ID. If `obj` is NULL, then `name` is a d-ptr to a C string. If `obj` is *not* NULL, then `name` is a p-ptr with respect to `obj`. The `fn` argument is a name resolver function to call (or NULL), whether a d-ptr or a p-ptr following the same rules as `name`. The flags argument is passed through to the resolving function.

If `fn` is NULL, then use the Twizzler Default Name Resolver. This uses the `TWZNAME` envvar to determine which object contains the system's default name object, and looks up the name in that.

The Default resolver ignores the `flags` argument.

Return Value

The resulting object ID is returned in `id` and the function returns 0 on success. On error, an error code is returned. If the called name resolver returns an error, that error is passed through.

Errors

Error codes include any error from the `fn`, but also: * `-EINVAL`: Invalid argument * `-ENOENT`: Name not found * `-EFAULT`: Could not load pointers either `name` or `fn` from `obj`.

twz_name_assign

```
#include <twz/name.h>
int twz_name_assign(objid_t id, const char *name);
```

Assign `name` to `id` in the Twizzler Default Name Resolver. Multiple names for an ID are allowed.

Return Value

Returns 0 on success, error code on error.

Errors

- `-ENOENT`: Could not find a name object.
- `-ENOMEM`: Not enough memory to assign name.

Twizzler System Calls (us/syscalls.md)

WARNING - these have recently changed; this documentation is pending an update.

All pointers passed to Twizzler system calls are d-ptrs (see Pointer Manipulation).

sys_ocreate

```
#include <twz/sys.h>
int sys_ocreate(int flags, objid_t kuid, objid_t src, objid_t *id);
```

Creates a new object, optionally copying from a source object. If `src` is zero, the new object will be zeroed (except for the meta-data page). The default metainfo structure will be generated based on the specified flags and the kuid. If copying from a source object, the meta-data page will contain the same data as the source object, except for the metainfo data controlled by the `flags` argument.

Valid bits for the `flags` argument are:

- TWZ_OC_HASHDATA: Hash the object data as part of ID generation. This flag sets MIP_HASHDATA in the p_flags entry of the metainfo structure. Cannot be used in conjunction with TWZ_OC_DFL_WRITE.
- TWZ_OC_ZERONONCE: Set the nonce to zero (do not randomly generate a nonce). This sets the nonce field of the metainfo structure to zero. Can only be used with TWZ_OC_HASHDATA.
- TWZ_OC_DFL_*: Where * is WRITE, READ, EXEC, or USE. Sets the default permissions for the new object, setting the associated MIP_DFL_* flag in the p_flags entry of the metainfo structure.
- TWZ_OC_PERSIST: Mark the object as persistent.

If src is non-zero, both objects will be set to copy-on-write and will share underlying pages until necessary to copy them. Objects used as source which are undergoing modification require a synchronization event (locking, atomic release, etc.) to ensure consistency between updates and copying for object creation.

Return Value

Returns 0 on success, and an error code on error. On success, the new object's ID is written into the id argument. The id argument is unchanged on failure.

Errors

- -EINVAL: Invalid argument.
- -EFAULT: id is not a valid pointer.
- -ENOENT: The src argument is non-zero and the object cannot be found.
- -ENOMEM: Not enough memory to fulfill the request.
- -ENOSPC: Object is persistent and there is not enough persistent storage.

sys_odelete

```
#include <twz/sys.h>
int sys_odelete(int flags, objid_t id);
```

Deletes the object specified by id. New accesses to the object will not be allowed (but current accesses will be allowed to continue). For example, if the object is mapped in a security context, access will still be allowed to continue. Once all in-kernel reference counts to the object reach zero, the object's resources will be freed.

The flags argument is for future expansion, and must currently be set to 0.

Return Value

Returns 0 on success, and an error code on error.

Errors

- -EINVAL: Invalid argument.
- -ENOENT: Could not find object id.

sys_ocfg

```
#include <twz/sys.h>
int sys_ocfg(int flags, objid_t id, long cmd, long arg);
```

Configure object id. The flags argument is for future expansion and must be set to 0. Valid commands (given by the cmd argument) are:

- TWZ_OCFG_PERSIST: Mark object as persistent (arg = 1) or volatile (arg = 0).

Return Value

Returns 0 on success, error code on error.

Errors

Generic error codes

- `-EINVAL`: Invalid argument.
- `-ENOENT`: Could not find object `id`.

`TWZ_OCFG_PERSIST`

- `-ENOSPC`: Not enough persistent storage to fulfill request.
- `-ENOMEM`: Not enough memory to fulfill request.

sys_attach

```
#include <twz/sys.h>
int sys_attach(objid_t parent, objid_t child, int flags, int type);
```

Attach a child KSO to a parent KSO. The parent KSO must already by a functioning KSO, whereas the child need not be. When attaching, the kernel will consider the child to be a KSO of type `type` (see Kernel State Objects). If the child is already considered a KSO by the kernel, and it has a different type than specified by `type`, the call will fail. The `flags` argument is for future expansion and must be set to 0.

If `parent` is zero, the call has the same effect as if made with `parent` being the ID of the thread's control object KSO.

The effect of this call depends on the type of KSO of the child:

- `KSO_ROOT`: Invalid; cannot attach a KSO root object to anything.
- `KSO_VIEW`: Changes a thread's view. Parent must be `KSO_THREAD`.
- `KSO_THREAD`: When attaching to a KSO root, this has the effect of starting a thread in the by invoking the `FAULT_RESUME` handler in the new thread as specified by the thread's object. The view of the new thread will be set to view specified in the thread's control object. `parent` must be a `KSO_ROOT`.
- `KSO_SECCTX`: Attaches a security context to a thread. `parent` must be `KSO_THREAD`.

Return Value

Returns 0 on success, error code on error.

Errors

- `-EINVAL`: Invalid argument.
- `-ENOENT`: Either child or parent could not be found.
- `-EBUSY`: Child has a different KSO type than specified.
- `-ENOMEM`: Not enough memory to fulfill request.

sys_detach

```
#include <twz/sys.h>
int sys_detach(objid_t parent, objid_t child, int flags, int type);
```

Detach a KSO child from its parent. Unlike `sys_attach`, this function allows delaying the actual detachment until some time in the future. The `type` argument must match the KSO type of the child. Similar to attach, if `parent` is zero, it treats the call as if it was made with `parent` being the thread control object of the current thread.

The `flags` argument specifies when detachment occurs, and *cannot* be zero. It is built from the following bits:

- `TWZ_DETACH_ONSYSCALL(s)`: Detach on system call number `s` (eg. `SYS_BECOME`). This is required.
- `TWZ_DETACH_REATTACH`: Detach on entry to the system call specified, and reattach on exit. Cannot be used in conjunction with any flags below.
- `TWZ_DETACH_ONENTRY`: Detach on entry to the specified system call.
- `TWZ_DETACH_ONEXIT`: Detach on exit from the specified system call.

Note that multiple calls to detach will *not* change previous detach specifications; they can only add. For example, the following calls,

```
sys_detach(0, S, TWZ_DETACH_ONSYSCALL(SYS_BECOME) | TWZ_DETACH_ONENTRY);
sys_detach(0, S, TWZ_DETACH_ONSYSCALL(SYS_BECOME) | TWZ_DETACH_REATTACH);
sys_detach(0, S, TWZ_DETACH_ONSYSCALL(SYS_BECOME) | TWZ_DETACH_ONEXIT);
```

will treat the second call as essentially a no-op. The third call will also have no effect, since the `ONENTRY` status of detachment remains across all three calls.

Return Value

Returns 0 on success, error code on error.

Errors

- `-EINVAL`: Invalid argument.
- `-ENOENT`: Either child or parent could not be found.
- `-EBUSY`: Child has a different KSO type than specified.
- `-ENOMEM`: Not enough memory to fulfill request.

sys_invalidate

```
#include <twz/sys.h>
int sys_invalidate(struct sys_invalidate_op ops[], size_t count);
```

Process a number `count` invalidation requests defined by the array `ops`. The array must have at least as many entries as `count`. An invalidation request will force the kernel to update any cached state is has derived from the bytes within the range of an object specified for an invalidation request. For example, updating a view entry may require the kernel to reconstruct page-tables. This system call is used to tell the kernel to do so.

The `sys_invalidate_op` structure has the following layout:

```
struct sys_invalidate_op {
    objid_t id;
    uint64_t offset;
    uint32_t length;
    uint16_t flags;
    uint16_t result;
};
```

where `id` specifies the ID of the KSO to invalidate, and `offset` and `length` describe the region of the KSO object that has changed. After processing an entry, the kernel writes the result into the `result` field.

The `flags` field can have the following bits set:

- `KSOI_VALID`: The entry is valid. This flag must be set for an entry to be processed.
- `KSOI_CURRENT`: Instead of looking up an object by the `id` field, interpret the `id` field as follows:
  - `KSO_CURRENT_VIEW`: Act as if the `id` were the current view of the current thread.
  - `KSO_CURRENT_ATTACHED_SECCTXS`: Act as if this call was made with all attached security contexts being invalidated with this range.

- KSO_CURRENT_THREAD: Act as if the id were the thread control object of the current thread.

Return Value

Returns the number of successful invalidations, or an error code. The result field is set to 0 on success or an error code for each valid invalidation operation entry.

Errors

The call itself can return the following errors:

- -EINVAL: Invalid argument.
- -EFAULT: The address of ops was invalid.

The result field in each valid op can take the following error codes:

- -EINVAL: Invalid argument.
- -ENOENT: Object id could not be found.
- -ENOMEM: Not enough memory.
- -EFAULT: Range referred to bytes outside the object.

sys_thrd_sync

```
#include <twz/sys.h>
int sys_thrd_sync(struct sys_thrd_sync_op ops[], size_t count);
```

Perform a synchronization operation on a number of sync addresses. A sync address is an address used by multiple threads (and/or the kernel) to synchronize on. Threads may either sleep on a sync address or wake sleeping threads on a sync address. This call operates on multiple sync addresses at once, allowing operations to be specified for each one.

The sys_thrd_sync_op structure has the following format:

```
struct sys_thrd_sync_op {
    long *addr;
    long arg;
    long res;
    uint32_t op;
    uint32_t flags;
    struct timespec timeout;
};
```

Each entry in ops refers to one operation on a sync address. The operation is specified by the op entry, and can be:

- THREAD_SYNC_SLEEP: Compare the value at *addr with arg. If they are the same, sleep waiting for a wakeup event on addr. If they differ, write THREAD_SYNC_RES_DIFFER into res and do not sleep. If flags has the THREAD_SYNC_OPT_TIMEOUT flag set, then sleep for a time specified by timeout, or until a wakeup event. If wakeup occurs because of timeout, res will contain THREAD_SYNC_RES_TIMEOUT instead.
- THREAD_SYNC_WAKE: Perform a wakeup event on addr, waking up any threads sleeping there. Wake up at most arg threads. If arg is negative, wake up all threads. The timeout field is ignored. Write the number of threads actually woken up into res.

The flags field can contain:

- THREAD_SYNC_OPT_VALID: Required. Marks an entry as valid.
- THREAD_SYNC_OPT_TIMEOUT: Indicates that the timeout field is valid, and a timeout should be applied.

Comparisons, sleeping, and wakeups are done atomically.

This function can specify multiple sync addresses at once. Waiting on multiple addresses will only sleep the thread if *all* sleep operations sleep. If only one of them does not, the thread will return immediately. Specifying multiple operations where some are sleeping and some are wakeups results in these operations being executed as expected, but in an unspecified order. Specifying a sleep and a wakeup for the same sync address in one call leads to unspecified behavior.

Return Value

Returns 0 on success, error code on error. The result of an operation is written to the `res` field for that operation, and contains one of the following depending on the operation.

`THREAD_SYNC_SLEEP`

- 0: The thread slept on this sync point, and was not woken up by it.
- `THREAD_SYNC_RES_WAKEUP`: The thread slept on this sync point, and was woken up by it.
- `THREAD_SYNC_RES_DIFFER`: The value at `*addr` was different from the value of `arg`.
- `THREAD_SYNC_RES_TIMEOUT`: A timeout occurred on this operation.
- An error code (negative).

Note that the kernel is only required to set *one* of the `res` fields for all the operations passed to it, though it might set more. If the thread was woken spuriously, no fields will be updated, and the function will return -EINTR.

`THREAD_SYNC_WAKE`

- The number of threads woken up (may be 0).
- An error code (negative).

Errors

Error codes for the function itself:

- `-EINVAL`: Invalid argument.
- `-EFAULT`: `ops` is an invalid pointer.
- `-EINTR`: Thread was interrupted.

Error codes for the `res` field:

- `-EINVAL`: Invalid argument.
- `-EFAULT`: `addr` is an invalid pointer.

sys_thrd_spawn

```
#include <twz/sys.h>
int sys_thrd_spawn(objid_t tid, struct sys_thrd_spawn_args *args, int flags);
```

Spawn a new kernel-thread using `tid` as the thread control object. The thread's initial state is specified by `args`. The `flags` argument is for future expansion and must be set to 0.

The `sys_thrd_spawn_args` structure has the following format:

```
struct sys_thrd_spawn_args {
    objid_t target_view;
    void (*start_func)(void *);
    void *arg;
    char *stack_base;
    size_t stack_size;
    char *tls_base;
```

```
    size_t thrd_ctrl;
};
```

The `target_view` field sets the new thread's view to the object specified. The new thread starts executing in the `start_func` function with `arg` as the argument. The stack pointer is set to an architecturally dependent value depending on `stack_base` and `stack_size` (eg. their sum, etc.). The TLS base for the thread is set to `tls_base`. The `thrd_ctrl` argument is for future expansion.

Return Value

Returns 0 on success, error code on error.

Errors

- `-EINVAL`: Invalid argument.
- `-ENOMEM`: Not enough memory to spawn thread.
- `-ENOENT`: The `tid` object or the `target_view` object could not be found.
- `-EFAULT`: One of the addresses in `args` referred to an invalid address, or the `args` argument was an invalid address.

sys_become

```
#include <twz/sys.h>
int sys_become(struct sys_become_args *args);
```

Set thread state to new state as specified by `args`, including instruction pointer, view (optionally), and registers. The new state is specified by sys_become_args, which is defined as follows:

```
struct sys_become_args {
    objid_t target_view;
    [architecture-specific contents; see Notes]
};
```

The object specified by `args->target_view` is set as the new view for the thread before it returns to userspace. If `args->target_view` is zero, the thread's view is unchanged.

Return Value

This function either does not return from the call-site, or it returns an error. A successful completion of this system call sets all registers to the state specified, thus this function only returns on failure.

Errors

- `-EINVAL`: `args` is an invalid pointer.
- `-EFAULT`: The thread would start executing in an invalid address.
- `-ENOENT`: The object specified by `args->target_view` does not exist, if `args->target_view` is non-zero.

Notes

The architecture-specific contents of sys_become_args is specified as follows:

x86_64:

```
struct sys_become_args {
    [architecture-independent contents]
    uint64_t target_rip;
```

```
    uint64_t rax;
    uint64_t rbx;
    uint64_t rcx;
    uint64_t rdx;
    uint64_t rdi;
    uint64_t rsi;
    uint64_t rsp;
    uint64_t rbp;
    uint64_t r8;
    uint64_t r9;
    uint64_t r10;
    uint64_t r11;
    uint64_t r12;
    uint64_t r13;
    uint64_t r14;
    uint64_t r15;
};
```

sys_thrd_ctl

```
#include <twz/sys.h>
int sys_thrd_ctl(int op, long arg);
```

Control kernel-thread related functions. The particular command is specified by `op`, as followed:

- `THRD_CTL_EXIT`: Exit the current thread. The `arg` argument, reinterpreted as `(long *)`, refers to a sync address (see `sys_thrd_sync`). If a valid address, the kernel will perform a wakeup event on all threads waiting on this address. If *not* a valid address, it is ignored. When specifying this operation, this function will *never* return and the kernel-thread will *always* exit.

There are additional commands depending on the architecture:

x86_64

- `THRD_CTL_SET_FS`: Set the user-space `fs` segment to the specified value in `arg`.
- `THRD_CTL_SET_GS`: Set the user-space `gs` segment to the specified value in `arg`.

Return Value

Returns 0 on success, error code on error. Not all operations return.

Errors

- `-EFAULT`: When `arg` refers to an address, and the address is an invalid address (except for 'THRD_CTL_EXIT).
- `-EINVAL`: Invalid argument.

Differences between kernel interface and C function wrappers

The actual interface to the kernel differs from that of the C function wrapper for some system calls. Most of the reason for this is that arguments are passed in registers, and object IDs are too big, thus they are split into high and low parts (denoted `_hi` and `_lo` respectively).

The actual interfaces are listed below:

```
int sys_ocreate(long kuid_lo, long kuid_hi, long src_lo, long src_hi,
                int flags, objid_t *id);
```

```
int sys_attach(long parent_lo, long parent_hi, long child_lo, long child_hi, long xflags);
```

The lower 32 bits of `xflags` contains the `type` argument, and the upper 32 bits contains the `flags` argument.

```
int sys_detach(long parent_lo, long parent_hi, long child_lo, long child_hi, long xflags);
```

The upper 32 bits of `xflags` contains the `flags` argument, minus the `TWZ_DETACH_ONSYSCALL(s)` data. The lower 32 bits contains the type and the `TWZ_DETACH_ONSYSCALL(s)` data.

```
int sys_thrd_spawn(long tid_lo, long tid_hi, struct sys_thrd_spawn_args *tsa, int flags);
```

Structured Objects

Most objects have a specific purpose, with a specific layout. Often, the object contains a *header*, usually located at the base of the data region of the object. An API over an object typically either assumes that the object has such a given layout, in which case the API might look like:

```
foo_action(twzobj *obj, ...);
```

However, this is inflexible. An object might contain sub-components that also have an API and a header. For example, many objects implement the events API (see Events API) as part of a larger structure (byte-streams, for example, are objects with a structure and thus a header struct in the base, but they *also* implement the events API which has its own header structure). Many APIs also provide variants of functions that do *not* assume their header struct is at the base. Such an API might provide a variant of the above function:

```
foo_action_hdr(twzobj *obj, struct foo_hdr *hdr, ...);
```

This allows an object to implement multiple APIs. The programmer must locate the appropriate header before passing it, of course. Often, within an object's API, this is straight-forward. For example, a header might look as follows:

```
struct bstream_hdr {
    ...
    struct event_hdr evh;
    ...
};
```

Then, within the bstream API, we can easily pass the event header to the event API functions we need to call.

However, not all headers can be located this way. An object might provide multiple APIs that *don't* depend on each other. For this, Twizzler provides functions to assist locating header structs, and registering and deregistering header structs. Each API that has a header that can be located this way with a *tag*. Each API that can be located this way provides a unique tag.

twz_object_getext

```
#include <twz/obj.h>
void *twz_object_getext(twzobj *obj, uint64_t tag);
```

Locate a structured object header within an object `obj` using `tag`, returning a d-ptr to the header structure associated with tag `tag`.

Return Value

Returns a pointer to the located header struct on success, NULL on failure. This function returns no errors, only returning NULL if the header struct was not found.

twz_object_addext

```
#include <twz/obj.h>
int twz_object_addext(twzobj *obj, uint64_t tag, void *ptr);
```

Add a pointer to a structured header to the list of headers that can be located with `twz_object_getext`. The pointer (`ptr`) must be a local pointer to object `obj`.

Return Value

Returns 0 on success, error code on error.

Errors

- `-EEXIST`: The tag already exists in the tag list.
- `-ENOMEM`: Not enough memory to add tag.

twz_object_delext

```
#include <twz/obj.h>
int twz_object_delext(twzobj *obj, uint64_t tag);
```

Remove a tag from the object's tag list.

Return Value

Returns 0 on success, error code on error.

Errors

- `-ENOENT`: `tag` was not found in the tag list.

Events API (us/events.md)

The event waiting and waking system functions in two parts:

- The waiter constructs a set of events upon which it waits, and then calls the wait function.
- The waker, when appropriate, wakes up all threads waiting on a particular event.

An object which implements the events API through structured objects can be found with the `EVENT_METAEXT_TAG`.

An events-enabled object will have a `struct evhdr` inside it which is used for waiting on events in that object. Every `struct evhdr` provides 64 unique events that can waited on, each a single bit in a 64-bit value. From here, the term "event" means a particular bit in a particular `evhdr`. If the bit is set, the event is "available", meaning that it has occurred, and something is ready. It will stay set until it is explicitly cleared (by `event_clear`). A thread which calls `event_wait` on an event that is not ready will (probably) sleep, whereas if it calls `event_wait` on an event which is active (bit set), the thread will not sleep. When waking a thread, the waker calls `event_wake`, which sets the specified bits (indicating the specified events are ready) and then wakes up waiting threads.

event_obj_init

```
#include <twz/event.h>
void event_obj_init(twzobj *obj, struct evhdr *hdr);
```

Initialize a `struct evhdr` within object `obj`. Must be called before any function that operates on the `hdr`.

event_init

```
#include <twz/event.h>
int event_init(struct event *ev, struct evhdr *hdr, uint64_t events, struct timespec *timeout);
```

Prepare a `struct event` (`ev`) for waiting. The `ev` argument will be filled out with the provided information and will be ready to be passwd `event_wake` function. The `events` argument is a bitfield of events that the caller wishes to wait on (eg. if `events` is 3, then we are waiting on event 1 and/or event 2). If the timeout field is NULL, then no timeout will be specified.

Return Value

Returns 0 on success, error code on error. ### Errors * `-EINVAL`: Invalid argument

event_wait

```
#include <twz/event.h>
int event_wait(size_t count, struct event *ev);
```

Wait on a set of events. The `count` argument describes the length of the array `ev`, which must contain a valid `struct event` (initialized via `event_init`) for each entry. The function will then wait for any specified event to occur.

The `struct event` has a result field (type `uint64_t`) which gets a copy of the available events for that object when the function returns.

This function can return spuriously.

Return Value

Returns the number of ready events on success, error code on error (negative).

Errors

This function can return any error specified by the `sys_thrd_sync` system call.

event_wake

```
#include <twz/event.h>
int event_wake(struct evhdr *ev, uint64_t events, long wcount);
```

Mark the events specified in `events` as occurred in `ev`, and `wcount` threads waiting for an event on this object. If wcount is -1, wakeup all threads. Note that there is no guarantee that the woken threads are *actually* waiting for the events specified in `events`, just that they are waiting for *an* event on this object. It is recommended to pass -1 for wcount unless you know what you're doing.

Return Value

Returns 0 on success, error code on error.

Errors

This function may return any error returned by `sys_thrd_sync`.

event_clear

```
#include <twz/event.h>
uint64_t event_clear(struct evhdr *hdr, uint64_t events);
```

Clear the events specified by `events` from `hdr`, returning any of those events if they are active. This function does not attempt to wake any threads, nor does it wait.

Return Value

Returns a bitwise-and of the pre-cleared events and the `events` argument, effectively returning which events were cleared. This function does not fail.

event_poll

```
#include <twz/event.h>
uint64_t event_poll(const struct evhdr *hdr, uint64_t events);
```

Return any active events specified by `events`. This function does not attempt to wake and threads, nor does it wait.

Return Value

Returns a bitwise-and of the triggered events in `hdr` and `events`. This function does not fail.

event_add_timeout

```
#include <twz/event.h>
void event_add_timeout(struct event *ev, const struct timespec *timeout);
```

Adds a timeout to the event specifier `ev`. The `timeout` argument must be non-NULL.

Twizzler IO Objects (us/io.md)

TWZIO_METAEXT_TAG

twzio_hdr

```
struct twzio_hdr {
    ssize_t (*read)(twzobj *, void *, size_t len, size_t off, unsigned flags);
    ssize_t (*write)(twzobj *, const void *, size_t len, size_t off, unsigned flags);
    int (*ioctl)(twzobj *, int request, long);
    int (*poll)(twzobj *, uint64_t type, struct event *event);
};
```

Events

TWZIO_EVENT_READ TWZIO_EVENT_WRITE TWZIO_EVENT_IOCTL

twzio_read, twzio_write

```
ssize_t twzio_read(twzobj *obj, void *buf, size_t len, size_t off, unsigned flags);

ssize_t twzio_write(twzobj *obj, const void *buf, size_t len, size_t off, unsigned flags);

ssize_t twzio_ioctl(twzobj *obj, int req, ...);
```

```
int twzio_poll(twzobj *, uint64_t, struct event *);
```

Valid flags include:

- `TWZIO_NONBLOCK`: Do not block if the operation cannot be performed immediately. If the function would have blocked, it returns `-EAGAIN`.

Faults and Fault Handling (us/faults.md)

Exceptions

Twizzler Security