NAME

tux − interact with the TUX kernel subsystem

SYNOPSIS

#include <sys/tuxmodule.h>

int tux (unsigned int action, user_req_t * req);

DESCRIPTION

The **tux()** system call calls the kernel to perform an *action* on behalf of the currently executing user-space TUX module.

action can be one of:

```
  enum tux_actions {
        TUX_ACTION_STARTUP = 1,
        TUX_ACTION_SHUTDOWN = 2,
        TUX_ACTION_STARTTHREAD = 3,
        TUX_ACTION_STOPTHREAD = 4,
        TUX_ACTION_EVENTLOOP = 5,
        TUX_ACTION_GET_OBJECT = 6,
        TUX_ACTION_SEND_OBJECT = 7,
        TUX_ACTION_READ_OBJECT = 8,
        TUX_ACTION_FINISH_REQ = 9,
        TUX_ACTION_FINISH_CLOSE_REQ = 10,
        TUX_ACTION_REGISTER_MODULE = 11,
        TUX_ACTION_UNREGISTER_MODULE = 12,
        TUX_ACTION_CURRENT_DATE = 13,
        TUX_ACTION_REGISTER_MIMETYPE = 14,
        TUX_ACTION_READ_HEADERS = 15,
        TUX_ACTION_POSTPONE_REQ = 16,
        TUX_ACTION_CONTINUE_REQ = 17,
        TUX_ACTION_REDIRECT_REQ = 18,
        TUX_ACTION_READ_POST_DATA = 19,
        TUX_ACTION_SEND_BUFFER = 20,
        TUX_ACTION_WATCH_PROXY_SOCKET = 21,
        TUX_ACTION_WAIT_PROXY_SOCKET = 22,
        TUX_ACTION_QUERY_VERSION = 23,
        MAX_TUX_ACTION
  };
```

The first *action* values listed below are administrative and are normally used only in the tux program.

TUX_ACTION_STARTUP starts the tux subsystem, and takes a NULL *req*. TODO: Only root can use TUX_ACTION_STARTUP.

TUX_ACTION_SHUTDOWN stops the tux subsystem, and takes any *req*, even a zero-filled *req*.

TUX_ACTION_STARTTHREAD is called once per thread with a *req->thread_nr* element monotonically increasing from 0.

TUX_ACTION_STOPTHREAD is not currently used by the tux daemon because all threads are automatically stopped on TUX_ACTION_SHUTDOWN. It remains available because it may be useful in circumstances that the tux daemon does not yet handle.

TUX_ACTION_REGISTER_MODULE Register a user-space module identified by the *req->modulename* string. One VFS name can be registered only once.

req->version_major, *req->version_minor*, and *req->version_patch* have to be set appropriately from TUX_MAJOR_VERSION, TUX_MINOR_VERSION, and TUX_PATCHLEVEL_VERSION, respectively; the kernel will sanity-check binary compatibility of the module.

TUX_ACTION_UNREGISTER_MODULE Unregister a user-space module identified by the req->modulename string. Only registered modules can be unregistered.

TUX_ACTION_CURRENT_DATE Set the current date string to req->new_date. The date string must be RFC 1123-compliant and increase monotonically. The tux daemon normally calls this once per second.

TUX_ACTION_REGISTER_MIMETYPE Sets the extension req->objectname to map to mimetype req->object_addr. The tux daemon normally registers the mime types in /etc/tux.mime.types, but modules could conceivably create their own mimetype mappings.

TUX_ACTION_QUERY_VERSION Return the major version, minor version, and patchlevel of the kernel TUX subsystem, encoded in the return value as

```
(TUX_MAJOR_VERSION << 24) | (TUX_MINOR_VERSION << 16) |
 TUX_PATCHLEVEL_VERSION
```

If the system call sets errno to EINVAL, assume major version 2, minor version 1.

The rest of the *action* values are used to respond to TUX events. The general architecture is that TUX's event loop is invoked to catch HTTP events, and then responses are generated in response to those events.

TUX_ACTION_EVENTLOOP invokes the TUX event loop—the TUX subsystem will either immediately return with a new request *req*, or will wait for new requests to arrive.

TUX_ACTION_GET_OBJECT issues a request for the URL object named in *req->objectname*. If the object is not immediately available then the currently handled request is suspended, and a new request is returned, or the TUX subsystem waits for new requests.

A URL object is a data stream that is accessed via a URL and is directly associated with a file pointed to by that URL. (In the future, we may extend the concept of a URL object.)

TUX_ACTION_SEND_OBJECT sends the current URL object to the client.

TUX_ACTION_READ_OBJECT reads the current URL object into the address specified by *req->object_addr*. TUX_ACTION_READ_OBJECT must not be called unless *req->objectlen* >= 0.

TUX_ACTION_READ_HEADERS reads a non-zero-delimited string into req->object_addr, with the length of the string kept in req->objectlen. This is a workaround used to read fields that tux does not currently parse; if you need it, report it as a bug so that more fields can be added to user_req (unless your use is so specialized that it will be of no general utility).

TUX_ACTION_POSTPONE_REQ postpones the request, meaning that no tux system calls will return data for this request until TUX_ACTION_CONTINUE_REQ is called.

TUX_ACTION_CONTINUE_REQ continues a postponed request. Unlike a normal TUX_ACTION, it takes as its argument the socket descriptor (this allows it to be called from a program that is unrelated to the program that called TUX_ACTION_POSTPONE_REQ if necessary). It is called like this:

```
   ret = tux(TUX_ACTION_CONTINUE_REQ, (user_req_t *)socket);
```

TUX_ACTION_READ_POST_DATA is an atomic action (it will always return with the same request, no need to handle a new request) that puts the non-zero-delimited POST data, up to the maximum set in req->objectlen (and limited by /proc/sys/net/tux/max_header_len), into req->object_addr, ands resets req->objectlen to the length.

TUX_ACTION_REDIRECT_REQ causes the request to be redirected to the secondary server. (No need to call TUX_ACTION_FINISH_REQ.)

TUX_ACTION_FINISH_REQ finishes and logs the request.

TUX_ACTION_FINISH_CLOSE_REQ is like TUX_ACTION_FINISH_REQ except that it also closes HTTP 1.1 keepalive connections.

TUX_ACTION_SEND_BUFFER is like TUX_ACTION_SEND_OBJECT except that it sends whatever is in the req->object_addr buffer. This can be used as a generic output buffer.

TUX_ACTION_WATCH_PROXY_SOCKET sets up a non-TUX socket to be used with TUX_ACTION_WAIT_PROXY_SOCKET. The socket must be a network socket. The function is atomic. Repeated calls to this action will replace the previous proxy socket, so there is no need to deinitialize it. The socket file descriptor must be put into req->object_addr.

TUX_ACTION_WAIT_PROXY_SOCKET postpones the current request until there are input packets on the socket that was set up via TUX_ACTION_WATCH_PROXY_SOCKET. The proxy socket has a keepalive timer running. The request will be resumed once there is input activity on the socket - the module can use nonblocking recv() on the socket to process input packets.

user_req_t *req* is the request returned by the TUX subsystem. Defined fields depend on the version. For major version 2, they are:

```
  typedef struct user_req_s {
        int version_major;
        int version_minor;
        int version_patch;

        int http_version;
        int http_method;

        int sock;
        int event;
        int thread_nr;
        void *id;
        void *priv;

        int http_status;
        int bytes_sent;
        char *object_addr;
        int module_index;
        char modulename[MAX_MODULENAME_LEN];

        unsigned int client_host;
        unsigned int objectlen;
        char query[MAX_URI_LEN];
        char objectname[MAX_URI_LEN];

        unsigned int cookies_len;
        char cookies[MAX_COOKIE_LEN];
```

```
            char content_type[MAX_FIELD_LEN];
            char user_agent[MAX_FIELD_LEN];
            char accept[MAX_FIELD_LEN];
            char accept_charset[MAX_FIELD_LEN];
            char accept_encoding[MAX_FIELD_LEN];
            char accept_language[MAX_FIELD_LEN];
            char cache_control[MAX_FIELD_LEN];
            char if_modified_since[MAX_FIELD_LEN];
            char negotiate[MAX_FIELD_LEN];
            char pragma[MAX_FIELD_LEN];
            char referer[MAX_FIELD_LEN];

            char *post_data;
            char new_date[DATE_LEN];
            int keep_alive;
    } user_req_t;
```

For major version 3, they are:

```
typedef struct user_req_s {
            uint32_t version_major;
            uint32_t version_minor;
            uint32_t version_patch;
            uint32_t http_version;
            uint32_t http_method;
            uint32_t http_status;

            uint32_t sock;
            uint32_t event;
            uint32_t error;
            uint32_t thread_nr;
            uint32_t bytes_sent;
            uint32_t client_host;
            uint32_t objectlen;
            uint32_t module_index;
            uint32_t keep_alive;
            uint32_t cookies_len;

            uint64_t id;
            uint64_t priv;
            uint64_t object_addr;

            uint8_t query[MAX_URI_LEN];
            uint8_t objectname[MAX_URI_LEN];
            uint8_t cookies[MAX_COOKIE_LEN];
            uint8_t content_type[MAX_FIELD_LEN];
            uint8_t user_agent[MAX_FIELD_LEN];
            uint8_t accept[MAX_FIELD_LEN];
            uint8_t accept_charset[MAX_FIELD_LEN];
            uint8_t accept_encoding[MAX_FIELD_LEN];
            uint8_t accept_language[MAX_FIELD_LEN];
            uint8_t cache_control[MAX_FIELD_LEN];
            uint8_t if_modified_since[MAX_FIELD_LEN];
            uint8_t negotiate[MAX_FIELD_LEN];
            uint8_t pragma[MAX_FIELD_LEN];
            uint8_t referer[MAX_FIELD_LEN];
```

```
        uint8_t new_date[DATE_LEN];
} user_req_t;
```

version_major
: Always set to TUX_MAJOR_VERSION, used to flag binary incompatibility.

version_minor
: Always set to TUX_MINOR_VERSION, used to flag binary incompatibility.

version_patch
: Always set to TUX_PATCHLEVEL_VERSION, used to flag binary incompatibility.

http_version
: One of **HTTP_1_0** or **HTTP_1_1**

http_method
: One of **METHOD_NONE**, **METHOD_GET**, **METHOD_HEAD**, **METHOD_POST**, or **METHOD_PUT**

sock
: Socket file descriptor; writing to this will send data to the connected client associated with this request. Do not read from this socket file descriptor; you could potentially confuse the HTTP engine.

event
: Private, per-request state for use in tux modules. The system will preserve this value as long as a request is active.

thread_nr
: Thread index; see discussion of *TUX_ACTION_STARTTHREAD*.

id
: A tux-daemon-internal value that is used to multiplex requests to the correct modules.

priv
: Works just like *event*, except that it is a pointer to private data instead of an integer.

http_status
: Set the error status as an integer for error reporting. The status is good by default, so it should not be modified except to report errors.

bytes_sent
: When you write to sock, you must set bytes_sent to the total number of bytes sent since the last tux() operation on this *req*, or the log entry's bytes sent counter will be incorrect. (This may change or disappear in future versions of tux.)

object_addr
: Set to an address for a buffer of at least *req->objectlen* size into which to read an object from the URL cache with the TUX_ACTION_READ_OBJECT *action*. TUX_ACTION_READ_OBJECT must not be called unless *req->objectlen* >= 0, and TUX implicitly relies on *req->object_addr* being at least *req->objectlen* in size.

module_index
: Used by the tux(8) daemon to determine which loadable module to associate with a *req*.

modulename
: The name of the module as set by TUX_ACTION_REGISTER_MODULE; private data to the tux daemon.

client_host
: The IP address of the host to which sock is connected.

objectlen
: The size of a file that satisfies the current request and which is currently living in the URL cache. This is set if a request returns after TUX_ACTION_GET_OBJECT. A module should make sure that the buffer at *req->object_addr* is at least *req->objectlen* in size before calling TUX_ACTION_READ_OBJECT.

query The full query string sent from the client.

objectname

Specifies the name of a URL to get with the TUX_ACTION_GET_OBJECT *action*. If the URL is not immediately available (that is, is not in the URL cache), the request is queued and the tux subsystem may go on to other ready requests while waiting.

cookies_len

If cookies are in the request header, *cookies_len* contains the length of the *cookies* string

cookies If cookies are in the request header, *cookies* is the string in which the cookies are passed to the module.

content_type

The Content-Type header value for the request

user_agent

The User-Agent header value for the request

accept The Accept header value for the request

accept_charset

The Accept-Charset header value for the request

accept_encoding

The Accept-Encoding header value for the request

accept_language

The Accept-Language header value for the request

cache_control

The Cache-Control header value for the request

if_modified_since

The If-Modified-Since header value for the request

negotiate

The Negotiate header value for the request

pragma The Pragma header value for the request

referer The Referer header value for the request

post_data

For POST requests, the incoming data is placed in post_data.

new_date

Returns the current date/time

keep_alive

The KeepAlive header value for the request

RETURN VALUE

tux() returns the following values:

```
enum tux_reactions {
        TUX_RETURN_USERSPACE_REQUEST = 0,
        TUX_RETURN_EXIT = 1,
        TUX_RETURN_SIGNAL = 2,
};
```

TUX_RETURN_USERSPACE_REQUEST means that the kernel has put a new request into *req*; the request must be responded to with one of TUX_ACTION_GET_OBJECT, TUX_ACTION_SEND_OBJECT, TUX_ACTION_READ_OBJECT, or TUX_ACTION_FINISH_REQ.

TUX_RETURN_EXIT means that TUX has been stopped.

TUX_RETURN_SIGNAL means that a signal has occured. No new request is scheduled.

ERRORS

Any negative value (such as -EFAULT, -EINVAL) is an indication of an error.

BUGS

This man page is incomplete.