Lthread Documentation

Release 1.0

Hasan Alayli

September 21, 2016

Contents

1 Introduction **1**
- 1.1 Installation . . . 2
- 1.2 Linking . . . 2
- 1.3 C++11 Bindings . . . 2
- 1.4 License . . . 2

2 Lthread **3**
- 2.1 lthread_create . . . 3
- 2.2 lthread_sleep . . . 3
- 2.3 lthread_cancel . . . 3
- 2.4 lthread_run . . . 4
- 2.5 lthread_join . . . 4
- 2.6 lthread_detach . . . 4
- 2.7 lthread_detach2 . . . 4
- 2.8 lthread_exit . . . 4
- 2.9 lthread_wakeup . . . 5
- 2.10 lthread_cond_create . . . 5
- 2.11 lthread_cond_wait . . . 5
- 2.12 lthread_cond_signal . . . 5
- 2.13 lthread_cond_broadcast . . . 6
- 2.14 lthread_set_data . . . 6
- 2.15 lthread_get_data . . . 6
- 2.16 lthread_current . . . 6
- 2.17 lthread_compute_begin . . . 6
- 2.18 lthread_compute_end . . . 7
- 2.19 DEFINE_LTHREAD . . . 7

3 Socket **9**
- 3.1 lthread_socket . . . 9
- 3.2 lthread_pipe . . . 9
- 3.3 lthread_accept . . . 9
- 3.4 lthread_close . . . 9
- 3.5 lthread_connect . . . 9
- 3.6 lthread_recv . . . 10
- 3.7 lthread_read . . . 10
- 3.8 lthread_readline . . . 10
- 3.9 lthread_recv_exact . . . 10
- 3.10 lthread_read_exact . . . 11

3.11 lthread_recvmsg . . . 11
3.12 lthread_recvfrom . . . 11
3.13 lthread_send . . . 11
3.14 lthread_write . . . 11
3.15 lthread_sendmsg . . . 11
3.16 lthread_sendto . . . 12
3.17 lthread_writev . . . 12
3.18 lthread_wait_read . . . 12
3.19 lthread_wait_write . . . 12
3.20 lthread_poll . . . 12

4 Disk IO **13**
4.1 lthread_io_read . . . 13
4.2 lthread_io_write . . . 13
4.3 lthread_sendfile . . . 13

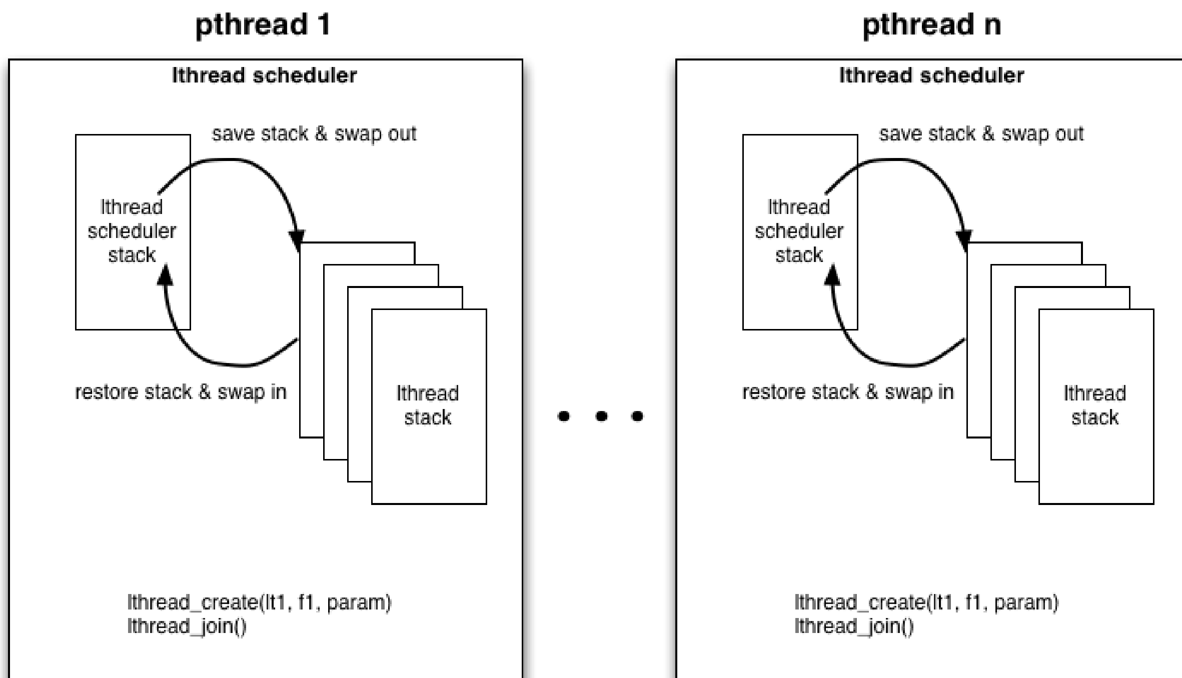
5 Examples **15**
5.1 Webserver . . . 15

CHAPTER 1

Introduction

lthread is a multicore/multithread coroutine library written in C. It uses [Sam Rushing's](https://github.com/samrushing) _swap function to swap lthreads. What's special about lthread is that it allows you to make *blocking calls* and *expensive* computations, blocking IO inside a coroutine, providing you with the advantages of coroutines and pthreads. See the http server example below.

lthreads are created in userspace and don't require kernel intervention, they are light weight and ideal for socket programming. Each lthread have separate stack, and the stack is madvise(2)-ed to save space, allowing you to create thousands(tested with a million lthreads) of coroutines and maintain a low memory footprint. The scheduler is hidden from the user and is created automagically in each pthread, allowing the user to take advantage of cpu cores and distribute the load by creating several pthreads, each running it's own lthread scheduler and handling its own share of coroutines. Locks are necessary when accessing global variables from lthreads running in different pthreads, and lthreads must not block on pthread condition variables as this will block the whole lthread scheduler in the pthread.



The lthread scheduler is created automagically in each pthread. Creating a new lthread using lthread_create() attaches the new lthread to the local scheduler in the pthread, and it will always run in the same scheduler for its lifetime.

To run an lthread scheduler in each pthread, launch a pthread and create lthreads using lthread_create() followed by

lthread_run() in each pthread.

Scheduler

The scheduler is build around epoll/kqueue and uses an rbtree to track which lthreads needs to run next.

If you need to execute an expensive computation or make a blocking call inside an lthread, you can surround the block of code with *lthread_compute_begin()* and *lthread_compute_end()*, which moves the lthread into an lthread_compute_scheduler that runs in its own pthread to avoid blocking other lthreads. lthread_compute_schedulers are created when needed and they die after 60 seconds of inactivity. *lthread_compute_begin()* tries to pick an already created and free lthread_compute_scheduler before it creates a new one.

1.1 Installation

Currently, lthread is supported on FreeBSD, OS X, and Linux (x86 & 64bit arch).

To build and install, simply:

```
cmake .
sudo make install
```

1.2 Linking

#include <lthread.h>

Pass *-llthread* to gcc to use lthread in your program.

1.3 C++11 Bindings

Docs and instructions to download/install can be found here.

1.4 License

Copyright (C) 2012, Hasan Alayli <halayli@gmail.com>

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER 2

Lthread

2.1 lthread_create

int **thread_create** (lthread_t ***new_lt*, lthread_func *func*, void **arg*)
Creates a new lthread.

Parameters

- **new_lt** (*lthread_t***) – a ptr->ptr to store the new lthread structure on success
- **func** (*lthread_func*) – Function to run in an lthread.
- **arg** (*void**) – Argument to pass to *func* when called.

Returns 0 on success with new_lt pointing to the new lthread.

Returns -1 on failure with *errno* specifying the reason.

lthread_func: void **(*)** (void*)

2.2 lthread_sleep

void **lthread_sleep** (uint64_t *msecs*)
Causes an lthread to sleep for *msecs* milliseconds.

Parameters

- **msecs** (*uint64_t*) – Number of milliseconds to sleep. *msecs=0* causes the lthread to yield and allow other lthreads to resume before it continues.

2.3 lthread_cancel

void **lthread_cancel** (lthread_t **lt*)
Cancels lthread and prepares it to be removed from lthread scheduler. If it was waiting for events, the events will get cancelled. If an lthread was joining on it, the lthread joining will get scheduled to run.

Parameters

- **lt** (*lthread_t**) – lthread to cancel

2.4 lthread_run

void **lthread_run** (void)
Runs lthread scheduler until all lthreads return.

2.5 lthread_join

int **lthread_join** (lthread_t **lt*, void ***ptr*, uint64_t *timeout*)
Blocks the calling lthread until lt has exited or a timeout occurred. In case of timeout, lthread_join returns -2 and lt doesn't get freed. If target lthread was cancelled, it returns -1 and the target lthread will be freed. **ptr will get populated by lthread_exit(). ptr cannot be from lthread's stack space. Joining on a joined lthread has undefined behavior.

Parameters

- **lt** (*lthread_t **) – lthread to join on.
- **ptr** (*void***) – optional, this ptr will be populated by `lthread_exit()`.
- **timeout** (*uint64_t*) – How long to wait trying to join on lt before timing out.

Returns 0 on success.

Returns -1 if target lthread got cancelled.

Returns -2 on timeout.

Attention: Joining on a joined lthread has undefined behavior

2.6 lthread_detach

void **lthread_detach** (void)
Marks the current lthread as detached, causing it to get freed once it exits. Otherwise `lthread_join()` must be called on the lthread to free it up. If an lthread wasn't marked as detached and wasn't joined on then a memory leak occurs.

2.7 lthread_detach2

void **lthread_detach2** (lthread_t **lt*)
Same as `lthread_detach()` except that it doesn't have to be called from within the lthread function. The lthread to detach is passed as a param.

Parameters

- **lt** (*lthread_t **) – Lthread to detach.

2.8 lthread_exit

void **lthread_exit** (void **ptr*)
Sets ptr value for the lthread calling `lthread_join()` and exits lthread.

Parameters

- **ptr** (*void**) – Optional, ptr value to pass to the joining lthread.

2.9 lthread_wakeup

void **lthread_wakeup**(lthread_t **lt*)

Wakes up a sleeping lthread. If lthread wasn't sleeping this function has no effect.

Parameters

- **lt** (*lthread_t **) – The lthread to wake up.

2.10 lthread_cond_create

int **lthread_cond_create**(lthread_cond_t ***c*)

Creates a condition variable that can be used between lthreads to block/signal each other.

Parameters

- **c** (*lthread_cond_t ***) – ptr->ptr that will be populated on success.

Returns 0 on success.

Returns -1 on error with *errno* containing the reason.

2.11 lthread_cond_wait

int **lthread_cond_wait**(lthread_cond_t **c*, uint64_t *timeout*)

Puts the lthread calling `lthread_cond_wait()` to sleep until *timeout* expires or another lthread signals it.

Parameters

- **c** (*lthread_cond_t **) – condition variable created by `lthread_cond_create()` and shared between lthreads requiring synchronization.
- **timeout** (*uint64_t*) – Number of milliseconds to wait on the condition variable to be signaled before it times out. 0 to wait indefinitely.

Returns 0 if it was signal.

Returns -2 on timeout.

2.12 lthread_cond_signal

void **lthread_cond_signal**(lthread_cond_t **c*)

Signals a single lthread blocked on `lthread_cond_wait()` to wake up and resume.

Parameters

- **c** (*lthread_cond_t **) – condition variable created by `lthread_cond_create()` and shared between lthreads requiring synchronization.

2.13 lthread_cond_broadcast

void **lthread_cond_broadcast** (lthread_cond_t **c*)

Signals all lthreads blocked on `lthread_cond_wait()` to wake up and resume.

Parameters

- **c** (*lthread_cond_t**) – condition variable created by `lthread_cond_create()` and shared between lthreads requiring synchronization.

2.14 lthread_set_data

void **lthread_set_data** (void **data*)

Sets data bound to the lthread. This value can be retrieved anywhere in the lthread using `lthread_get_data()`.

Parameters

- **data** (*void**) – value to be set.

2.15 lthread_get_data

void ***lthread_get_data** (void)

Returns the value set for the current lthread.

Returns Value set by `lthread_set_data()`

2.16 lthread_current

lthread_t ***lthread_current** ()

Returns a pointer to the current lthread.

Returns ptr to the current lthread running.

2.17 lthread_compute_begin

int **lthread_compute_begin** (void)

Resumes lthread inside a pthread to run expensive computations or make a blocking call like *gethostbyname()*. This call *must* be followed by `lthread_compute_end()` after the computation and/or blocking calls statements have been made, to resume the lthread in its original lthread scheduler. No lthread_* calls can be made during the 2 calls.

Returns 0 on success.

Returns -1 if lthread failed to resume it in a pthread.

2.18 lthread_compute_end

void **lthread_compute_end**(void)
Moves lthread from pthread back to the lthread scheduler it was running on.

2.19 DEFINE_LTHREAD

DEFINE_LTHREAD(name)
Sets the name of the function inside the lthread structure for easier crash debugging. Must be called inside the lthread.

CHAPTER 3

Socket

3.1 lthread_socket

int **lthread_socket** (int *domain*, int *type*, int *protocol*)
Creates a new socket(2) and sets it to non-blocking.

Parameters can be found in *man socket*.

3.2 lthread_pipe

int **lthread_pipe** (int *fildes[2]*)
lthread version of pipe(2), with socket set to non-blocking.

3.3 lthread_accept

int **lthread_accept** (int *fd*, struct sockaddr *, socklen_t *)
lthread version of accept(2). *man 2 accept* for more details.

3.4 lthread_close

int **lthread_close** (int *fd*)
lthread version of close(2). *man 2 close* for more details.

3.5 lthread_connect

int **lthread_connect** (int *fd*, struct sockaddr *, socklen_t, uint64_t *timeout*)
lthread version of connect(2) with additional timeout parameter.

Returns new fd > 0 on success.

Returns -1 on failure.

Returns -2 on timeout.

3.6 lthread_recv

ssize_t **lthread_recv** (int *fd*, void **buf*, size_t *buf_len*, int *flags*, uint64_t *timeout*)
lthread version of recv(2), with additional timeout parameter.

Returns Returns number of bytes read, -1 on failure and -2 on timeout.

3.7 lthread_read

ssize_t **lthread_read** (int *fd*, void **buf*, size_t *length*, uint64_t *timeout*)
lthread version of read(2), with additional timeout parameter.

Returns Returns number of bytes read.

Returns 0 if socket is closed.

Returns -1 on failure.

Returns -2 on timeout.

3.8 lthread_readline

ssize_t **lthread_readline** (int *fd*, char ***buf*, size_t *max*, uint64_t *timeout*)
Keeps reading from fd until it hits a \n or *max* bytes.

Parameters

- **fd** (*int*) – file descriptor.
- ****buf** (*char*) – Ptr->ptr that will contain the line read(must be freed).
- **max** (*size_t*) – Maximum number of bytes to read before finding \n.
- **timeout** – Milliseconds to wait on reading before timing out.

Returns Number of bytes read.

Returns 0 if socket is closed.

Returns -1 on failure.

Returns -2 on timeout.

3.9 lthread_recv_exact

ssize_t **lthread_recv_exact** (int *fd*, void **buf*, size_t *buf_len*, int *flags*, uint64_t *timeout*)
Blocks until exact number of bytes are read.

Returns Number of bytes read.

Returns 0 if socket is closed.

Returns -1 on failure.

Returns -2 on timeout.

3.10 lthread_read_exact

ssize_t **lthread_read_exact** (int *fd*, void **buf*, size_t *length*, uint64_t *timeout*)

Blocks until exact number of bytes are read.

Returns Number of bytes read.

Returns 0 if socket is closed.

Returns -1 on failure.

Returns -2 on timeout.

3.11 lthread_recvmsg

ssize_t **lthread_recvmsg** (int *fd*, struct msghdr **message*, int *flags*, uint64_t *timeout*)

lthread version of recvmsg(2). *man 2 recvmsg* for more details.

Returns Returns number of bytes read, -1 on failure and -2 on timeout.

3.12 lthread_recvfrom

ssize_t **lthread_recvfrom** (int *fd*, void **buf*, size_t *length*, int *flags*, struct sockaddr **address*, socklen_t **address_len*, uint64_t *timeout*)

lthread version of recvfrom(2). *man 2 recvfrom* for more details.

Returns Returns number of bytes read.

Returns 0 if socket is closed.

Returns -1 on failure.

Returns -2 on timeout.

3.13 lthread_send

ssize_t **lthread_send** (int *fd*, const void **buf*, size_t *buf_len*, int *flags*)

lthread version of send(2). *man 2 send* for more details.

3.14 lthread_write

ssize_t **lthread_write** (int *fd*, const void **buf*, size_t *buf_len*)

lthread version of write(2). *man 2 write* for more details.

3.15 lthread_sendmsg

ssize_t **lthread_sendmsg** (int *fd*, const struct msghdr **message*, int *flags*)

lthread version of sendmsg(2). *man 2 sendmsg* for more details.

3.16 lthread_sendto

ssize_t **lthread_sendto** (int *fd*, const void **buf*, size_t *length*, int *flags*, const struct sockaddr **dest_addr*, socklen_t *dest_len*)

lthread version of sendto(2). *man 2 sendto* for more details.

3.17 lthread_writev

ssize_t **lthread_writev** (int *fd*, struct iovec **iov*, int *iovcnt*)

lthread version of writev(2). *man 2 writev* for more details.

3.18 lthread_wait_read

int **lthread_wait_read** (int *fd*, int *timeout_ms*)

Waits for an fd to become readable.

Returns 0 on success.

Returns -2 on timeout.

3.19 lthread_wait_write

int **lthread_wait_write** (int *fd*, int *timeout_ms*)

Waits for an fd to become writable.

Returns 0 on success.

Returns -2 on timeout.

3.20 lthread_poll

int **lthread_poll** (struct pollfd **fds*, nfds_t *nfds*, int *timeout*)

Lthread version of poll(2)

Returns -1 on poll(2) error (when timeout == 0).

Returns 0 on timeout.

Returns > 0 to indicate the # of fds returned.

Note: If timeout == 0, poll(2) is called directly and the lthread never goes to sleep.

CHAPTER 4

Disk IO

The way disk async functions are implemented in lthread is by using a native worker thread in the background to execute the actual read/write calls to disk. When an lthread calls `lthread_io_read()` or `lthread_io_write()` a job is put on a queue for the native thread to pick up and the actual lthread yields until the read/write is done.

Use `lthread_io_read()` or `lthread_io_write()` when fd is a file descriptor to a file.

4.1 lthread_io_read

ssize_t **lthread_io_read**(int *fd*, void **buf*, size_t *nbytes*)
An async version of read(2) for disk IO.

4.2 lthread_io_write

ssize_t **lthread_io_write**(int *fd*, const void **buf*, size_t *buf_len*)
An async version of write(2) for disk IO.

4.3 lthread_sendfile

int **lthread_sendfile**(int *fd*, int *s*, off_t *offset*, size_t *nbytes*, struct sf_hdtr **hdtr*)
An lthread version of sendfile(2). *man 2 sendfile* for more details.

Note: Available on FreeBSD only.

CHAPTER 5

Examples

5.1 Webserver

gcc -I/usr/local/include -llthread test.c -o test

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <lthread.h>

struct cli_info {
    /* other stuff if needed*/
    struct sockaddr_in peer_addr;
    int fd;
};

typedef struct cli_info cli_info_t;

char *reply = "HTTP/1.0 200 OK\r\nContent-length: 11\r\n\r\nHello World";

unsigned int
fibonacci(unsigned int n)
{
    if (n == 0)
        return 0;
     if (n == 1)
        return 1;

     return fibonacci(n - 1) + fibonacci(n - 2);
}

void
http_serv(lthread_t *lt, void *arg)
{
    cli_info_t *cli_info = arg;
    char *buf = NULL;
    int ret = 0;
    char ipstr[INET6_ADDRSTRLEN];
    lthread_detach();

    inet_ntop(AF_INET, &cli_info->peer_addr.sin_addr, ipstr, INET_ADDRSTRLEN);
    printf("Accepted connection on IP %s\n", ipstr);
```

```
    if ((buf = malloc(1024)) == NULL)
        return;

    /* read data from client or timeout in 5 secs */
    ret = lthread_recv(cli_info->fd, buf, 1024, 0, 5000);

    /* did we timeout before the user has sent us anything? */
    if (ret == -2) {
        lthread_close(cli_info->fd);
        free(buf);
        free(arg);
        return;
    }

    /*
     * Run an expensive computation without blocking other lthreads.
     * lthread_compute_begin() will yield http_serv coroutine and resumes
     * it in a compute scheduler that runs in a pthread. If a compute scheduler
     * is already available and free it will be used otherwise a compute scheduler
     * is created and launched in a new pthread. After the compute scheduler
     * resumes the lthread it will wait 60 seconds for a new job and dies after 60
     * of inactivity.
     */
    lthread_compute_begin();
        /* make an expensive call without blocking other coroutines */
        ret = fibonacci(35);
    lthread_compute_end();

    /* reply back to user */
    lthread_send(cli_info->fd, reply, strlen(reply), 0);
    lthread_close(cli_info->fd);
    free(buf);
    free(arg);
}

void
listener(void *arg)
{
    int cli_fd = 0;
    int lsn_fd = 0;
    int opt = 1;
    int ret = 0;
    struct sockaddr_in peer_addr = {};
    struct   sockaddr_in sin = {};
    socklen_t addrlen = sizeof(peer_addr);
    lthread_t *cli_lt = NULL;
    cli_info_t *cli_info = NULL;
    char ipstr[INET6_ADDRSTRLEN];
    lthread_detach();

    DEFINE_LTHREAD;

    /* create listening socket */
    lsn_fd = lthread_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (lsn_fd == -1)
        return;

    if (setsockopt(lsn_fd, SOL_SOCKET, SO_REUSEADDR, &opt,sizeof(int)) == -1)
```

```
        perror("failed to set SOREUSEADDR on socket");

    sin.sin_family = PF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(3128);

    /* bind to the listening port */
    ret = bind(lsn_fd, (struct sockaddr *)&sin, sizeof(sin));
    if (ret == -1) {
        perror("Failed to bind on port 3128");
        return;
    }

    printf("Starting listener on 3128\n");

    listen(lsn_fd, 128);

    while (1) {
        /* block until a new connection arrives */
        cli_fd = lthread_accept(lsn_fd, (struct sockaddr*)&peer_addr, &addrlen);
        if (cli_fd == -1) {
            perror("Failed to accept connection");
            return;
        }

        if ((cli_info = malloc(sizeof(cli_info_t))) == NULL) {
            close(cli_fd);
            continue;
        }
        cli_info->peer_addr = peer_addr;
        cli_info->fd = cli_fd;
        /* launch a new lthread that takes care of this client */
        ret = lthread_create(&cli_lt, http_serv, cli_info);
    }
}

int
main(int argc, char **argv)
{
    lthread_t *lt = NULL;

    lthread_create(&lt, listener, NULL);
    lthread_run();

    return 0;
}
```

(C type), 3

D

DEFINE_LTHREAD (C macro), 7

L

lthread_accept (C function), 9
lthread_cancel (C function), 3
lthread_close (C function), 9
lthread_compute_begin (C function), 6
lthread_compute_end (C function), 7
lthread_cond_broadcast (C function), 6
lthread_cond_create (C function), 5
lthread_cond_signal (C function), 5
lthread_cond_wait (C function), 5
lthread_connect (C function), 9
lthread_current (C function), 6
lthread_detach (C function), 4
lthread_detach2 (C function), 4
lthread_exit (C function), 4
lthread_get_data (C function), 6
lthread_io_read (C function), 13
lthread_io_write (C function), 13
lthread_join (C function), 4
lthread_pipe (C function), 9
lthread_poll (C function), 12
lthread_read (C function), 10
lthread_read_exact (C function), 11
lthread_readline (C function), 10
lthread_recv (C function), 10
lthread_recv_exact (C function), 10
lthread_recvfrom (C function), 11
lthread_recvmsg (C function), 11
lthread_run (C function), 4
lthread_send (C function), 11
lthread_sendfile (C function), 13
lthread_sendmsg (C function), 11
lthread_sendto (C function), 12
lthread_set_data (C function), 6
lthread_sleep (C function), 3
lthread_socket (C function), 9
lthread_wait_read (C function), 12
lthread_wait_write (C function), 12
lthread_wakeup (C function), 5
lthread_write (C function), 11
lthread_writev (C function), 12

T

thread_create (C function), 3