Ejemplo de cómo la biblioteca LinuxThreads crea una hebra con la función pthread_create()

A) Función `pthread_create()`:

Los pasos indicados en negrita realizan lo siguiente:

1. Se crea un cauce (pipe) para comunicar la hebra que invoca la función con la hebra gestora de la biblioteca.
2. Se crea la hebra gestora.
3. Se escribe en el cauce la petición de creación de hebra junto con los argumentos para su creación.

Archivo de la biblioteca: **pthread.c**

```
/* Linuxthreads - a simple clone()-based implementation of Posix        */
/* threads for Linux.                                                   */
/* Copyright (C) 1996 Xavier Leroy (Xavier.Leroy@inria.fr)              */
/*                                                                      */
/* This program is free software; you can redistribute it and/or        */
/* modify it under the terms of the GNU Library General Public License  */
/* as published by the Free Software Foundation; either version 2       */
/* of the License, or (at your option) any later version.               */
. . .
/* Thread creation, initialization, and basic low-level routines */

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "pthread.h"
#include "internals.h"
#include "spinlock.h"
#include "restart.h"

/* Descriptor of the initial thread */

. . .

int __pthread_manager_request = -1;

/* Other end of the pipe for sending requests to the thread manager. */

int __pthread_manager_reader;

. . .

static int pthread_initialize_manager(void)
{
  int manager_pipe[2];

  /* If basic initialization not done yet (e.g. we're called from a
```

```
     constructor run before our constructor), do it now */
  if (__pthread_initial_thread_bos == NULL) pthread_initialize();
  /* Setup stack for thread manager */
  __pthread_manager_thread_bos = malloc(THREAD_MANAGER_STACK_SIZE);
  if (__pthread_manager_thread_bos == NULL) return -1;
  __pthread_manager_thread_tos =
    __pthread_manager_thread_bos + THREAD_MANAGER_STACK_SIZE;
  /* Setup pipe to communicate with thread manager */
  if (pipe(manager_pipe) == -1) {
    free(__pthread_manager_thread_bos);
    return -1;
  }
  __pthread_manager_request = manager_pipe[1]; /* writing end */
  __pthread_manager_reader = manager_pipe[0]; /* reading end */
  /* Start the thread manager */
  __pthread_manager_pid =
    __clone(__pthread_manager, (void **) __pthread_manager_thread_tos,
            CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND,
            (void *) manager_pipe[0]);
  if (__pthread_manager_pid == -1) {
    free(__pthread_manager_thread_bos);
    close(manager_pipe[0]);
    close(manager_pipe[1]);
    __pthread_manager_request = -1;
    return -1;
  }
  return 0;
}

/* Thread creation */

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void * (*start_routine)(void *), void *arg)
{
  pthread_descr self = thread_self();
  struct pthread_request request;
  if (__pthread_manager_request < 0) {
    if (pthread_initialize_manager() < 0) return EAGAIN;
  }
  request.req_thread = self;
  request.req_kind = REQ_CREATE;
  request.req_args.create.attr = attr;
  request.req_args.create.fn = start_routine;
  request.req_args.create.arg = arg;
  sigprocmask(SIG_SETMASK, (const sigset_t *) NULL,
              &request.req_args.create.mask);
  write(__pthread_manager_request, (char *) &request, sizeof(request));
  suspend(self);
  if (self->p_retcode == 0) *thread = (pthread_t) self->p_retval;
  return self->p_retcode;
}

/* Simple operations on thread identifiers */

pthread_t pthread_self(void)
{
  pthread_descr self = thread_self();
  return self->p_tid;
}

. . .
```

B) Implementación del gestor de hebras

En el código resaltado, se muestra como:

1. La hebra gestora sondea los descriptores se cauces abiertos con select()
2. Cuando detecta que se ha escrito una petición en un cauce, determina su tipo y pasa al punto 3.
3. Si la petición es de creación, invoca a la llamada al sistema clone con los indicadores **CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND.**

Archivo: **manager.c**

```
/* Linuxthreads - a simple clone()-based implementation of Posix        */
/* threads for Linux.                                                   */
/* Copyright (C) 1996 Xavier Leroy (Xavier.Leroy@inria.fr)              */
/*                                                                      */
/* This program is free software; you can redistribute it and/or        */
/* modify it under the terms of the GNU Library General Public License  */
/* as published by the Free Software Foundation; either version 2       */
/* of the License, or (at your option) any later version.               */

. . .

/* The "thread manager" thread: manages creation and termination of threads */

#define _REENTRANT
#include <errno.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>           /* for select */
#include <sys/types.h>          /* for select */
#include <sys/mman.h>           /* for mmap */
#include <sys/wait.h>           /* for waitpid macros */
#include <linux/sched.h>

#include "pthread.h"
#include "internals.h"
#include "spinlock.h"
#include "restart.h"

/* Array of active threads. Entry 0 is reserved for the initial thread. */

struct pthread_handle_struct __pthread_handles[PTHREAD_THREADS_MAX] =
{ { 0, &__pthread_initial_thread}, /* All NULLs */ };

. . .
```

/* La hebra servidora gestiona las solicitudes de creación u terminación de hebras */

```
int __pthread_manager(void * arg)
{
  int reqfd = (int) arg;
  sigset_t mask;
  fd_set readfds;
  struct timeval timeout;
  int n;
```

```
  struct pthread_request request;

  /* If we have special thread_self processing, initialize it.  */
#ifdef INIT_THREAD_SELF
  INIT_THREAD_SELF(&__pthread_manager_thread);
#endif
  /* Block all signals except PTHREAD_SIG_RESTART */
  sigfillset(&mask);
  sigdelset(&mask, PTHREAD_SIG_RESTART);
  sigprocmask(SIG_SETMASK, &mask, NULL);
  /* Enter server loop */
  while(1) {
    FD_ZERO(&readfds);
    FD_SET(reqfd, &readfds);
    timeout.tv_sec = 2;
    timeout.tv_usec = 0;

/* La hebra gestora realiza un select() sobre el cauce para no realizar un
read() que la bloqueará si no hay información disponible */

    n = select(FD_SETSIZE, &readfds, NULL, NULL, &timeout);
    /* Check for termination of the main thread */
    if (getppid() == 1) {
      pthread_kill_all_threads(SIGKILL, 0);
      _exit(0);
    }
    /* Check for dead children */
    if (terminated_children) {
      terminated_children = 0;
      pthread_reap_children();
    }
    /* Read and execute request */
    if (n == 1 && FD_ISSET(reqfd, &readfds)) {
      n = read(reqfd, (char *)&request, sizeof(request));
      ASSERT(n == sizeof(request));
      switch(request.req_kind) {
      case REQ_CREATE:
        request.req_thread->p_retcode =
          pthread_handle_create((pthread_t *) &request.req_thread->p_retval,
                                request.req_args.create.attr,
                                request.req_args.create.fn,
                                request.req_args.create.arg,
                                &request.req_args.create.mask,
                                request.req_thread->p_pid);
        restart(request.req_thread);
        break;
      case REQ_FREE:

      . . .
      }
    }
  }
}

/* Process creation */

. . .

static int pthread_handle_create(pthread_t *thread, const pthread_attr_t *attr,
                                 void * (*start_routine)(void *), void *arg,
                                 sigset_t * mask, int father_pid)
```

```
{
  size_t sseg;
  int pid;
  pthread_descr new_thread;
  pthread_t new_thread_id;
  int i;

  /* Find a free stack segment for the current stack */
  for (sseg = 1; ; sseg++) {
    if (sseg >= PTHREAD_THREADS_MAX) return EAGAIN;
    if (__pthread_handles[sseg].h_descr != NULL) continue;
    new_thread = thread_segment(sseg);
    /* Allocate space for stack and thread descriptor. */
    if (mmap((caddr_t)((char *)(new_thread+1) - INITIAL_STACK_SIZE),
             INITIAL_STACK_SIZE, PROT_READ | PROT_WRITE | PROT_EXEC,
             MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED | MAP_GROWSDOWN, -1, 0)
        != (caddr_t) -1) break;
    /* It seems part of this segment is already mapped. Try the next. */
  }
  /* Allocate new thread identifier */
  pthread_threads_counter += PTHREAD_THREADS_MAX;
  new_thread_id = sseg + pthread_threads_counter;
  /* Initialize the thread descriptor */
  new_thread->p_nextwaiting = NULL;
  new_thread->p_tid = new_thread_id;
  new_thread->p_priority = 0;
  new_thread->p_spinlock = &(__pthread_handles[sseg].h_spinlock);
  new_thread->p_signal = 0;
  new_thread->p_signal_jmp = NULL;
  new_thread->p_cancel_jmp = NULL;
  new_thread->p_terminated = 0;
  new_thread->p_detached = attr == NULL ? 0 : attr->detachstate;
  new_thread->p_exited = 0;
  new_thread->p_retval = NULL;
  new_thread->p_joining = NULL;
  new_thread->p_cleanup = NULL;
  new_thread->p_cancelstate = PTHREAD_CANCEL_ENABLE;
  new_thread->p_canceltype = PTHREAD_CANCEL_DEFERRED;
  new_thread->p_canceled = 0;
  new_thread->p_errno = 0;
  new_thread->p_h_errno = 0;
  . . .
  new_thread->p_start_args.start_routine = start_routine;
  new_thread->p_start_args.arg = arg;
  new_thread->p_start_args.mask = *mask;
  /* Do the cloning */
  pid = __clone(pthread_start_thread, (void **) new_thread,
                CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND |
                PTHREAD_SIG_RESTART,
                new_thread);
  /* Check if cloning succeeded */
  if (pid == -1) {
    /* Free the stack */
. . .
}
```