Parte IV

Apéndices

Diseño y realización de un servidor de procesadores

"Conjunción del máximo de libertad con el máximo de orden. La libertad, visible, el orden oculto."

Carlos Pujol ("Cuaderno de escritura")

Apéndice A: Diseño y realización de un servidor de procesadores

A.1. Introducción 191
A.2. Descripción del interfaz 192
A.2.1. Cpu_server_request 194
A.2.2. Cpu_server_release 194
A.2.3. Cpu_server_info 194
A.3. Detalles del diseño sobre Mach 195
A.3.1. Identificación de las peticiones 195
A.3.2. Servidor multiflujo 195
A.3.3. Gestión de excepciones 195

A.1. Introducción

Para dar soporte a la posibilidad de una planificación basada en políticas de espacio compartido necesitamos una utilidad para asignar procesadores a aplicaciones y permitir así el particionado del multiprocesador y la coexistencia de varios modelos de programación. Las aplicaciones paralelas que desean sacar un alto rendimiento a la máquina necesitan saber cuántos procesadores están disponibles, cuántos pueden pedir en este instante, qué procesadores son, etc.

Presentamos en este apéndice la realización de un servidor de procesadores o CPU *server*, es decir, un proceso privilegiado (una task) que se encarga de gestionar los procesadores físicos en favor de las aplicaciones que se los pidan. Los clientes de este servidor son aplicaciones paralelas que corren sobre un multiprocesador de memoria compartida. Los clientes suelen ser una task con varios threads concurrentes (pero en la realización actual está abierto que pueda ser también una aplicación que incumba a varias task).

El interfaz de las llamadas al servidor es al principio muy sencilla; se compone únicamente de tres llamadas. La primera es para solicitar un determinado número de procesadores, la segunda es para devolverlos y la tercera para pedir información.

Mach ofrece la posibilidad de agrupar procesadores físicos y flujos de ejecución para que éstos últimos se ejecuten sobre los primeros; el objeto de kernel que los reúne es el *processor set* o grupo de procesadores.

La aplicación que necesite procesadores en exclusiva tiene que crear uno o varios *processor set* propios y a continuación pedir al servidor que le inserte procesadores en uno de esos *processor sets*.

La aplicación (conjunto de tasks) decide también qué threads se ejecutarán en cada grupo de procesadores, asignando y desasignando sus threads a sus *processor set*.

Un servidor de nombres llamado “snames” aporta la posibilidad de registrar *ports* de servicios, dándoles un nombre, de forma que las aplicaciones que necesiten acceder al servicio puedan preguntar por el puerto al que dirigir sus peticiones. Para programar las comunicaciones entre las aplicaciones y el servidor se ha utilizado el generador de llamadas a procedimiento remoto (RPC) incluido en Mach, el Mach Interface Generator (MIG).

La política de asignación de nuestro CPU *server* es actualmente muy sencilla. Asigna procesadores bajo petición, generalmente tantos como pida el usuario o, sino, los que estén disponibles en ese momento. Los libera, es decir, los devuelve al conjunto de procesadores de uso general, también tras una petición de la aplicación.

Ofrece tres RPC a sus clientes:

cpu_server_request(), para solicitar N procesadores, con varias opciones: los que haya disponibles; se quieren todos o ninguno; el servidor decide cuantos; necesitamos 2^k procesadores entre 1 y N, etc.

cpu_server_release(), para retornar N procesadores para uso general.

cpu_server_info(), para recabar información diversa sobre el estado de los procesadores y los grupos de procesadores controlados por el CPU *server*.

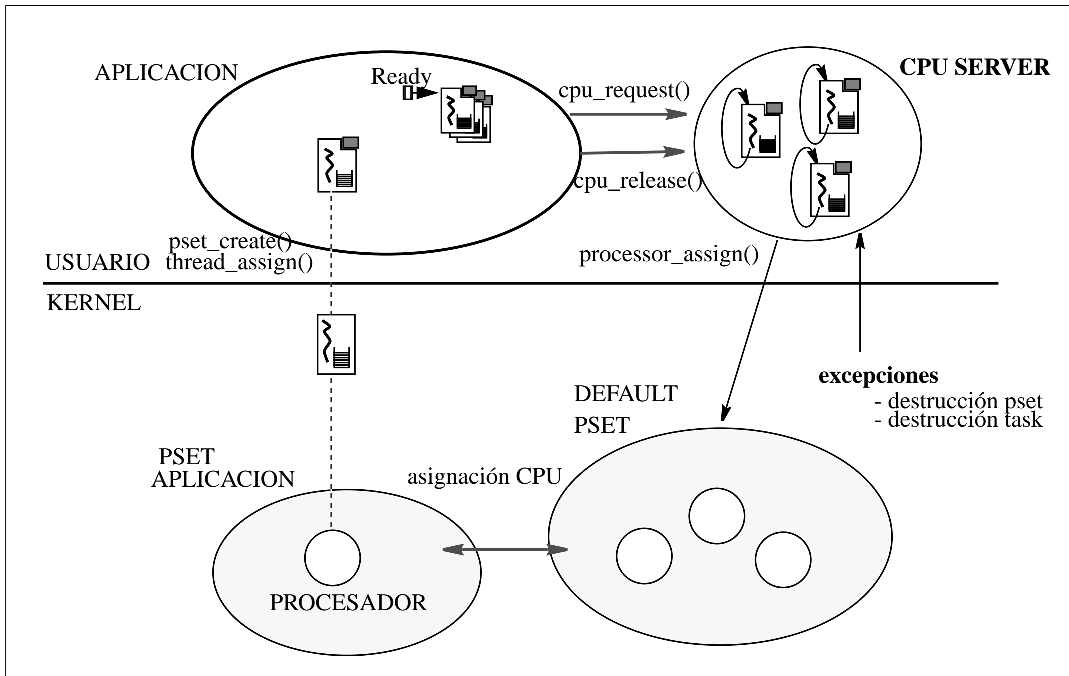

Figura A-1 Relación entre la aplicación, el CPU *server* y el kernel.

La Figura A-1 muestra la aplicación y el CPU *server*. La flechas indican las llamadas al kernel y los mensajes que intercambian las tasks. La llamada processor_assign(), que reasigna un procesador a un pset, es privilegiada y, por tanto, en la realización actual de Mach, sólo tiene efecto desde un proceso cuyo usuario efectivo sea "root", herencia clara de UNIX.

A.2. Descripción del interfaz

A continuación especificamos la interface de llamadas al servidor de procesadores. La aplicación que utilice estas llamadas debe montarse con la librería que contiene la parte de usuario de los RPC generados por MIG. La librería se llama libcpu.a y está en el directorio /usr/mach3/ccs/lib.

Ficheros de definiciones que ha de incluir la aplicación:

```
/*CPU CLIENT */

#include <mach.h>
#include <mach/message.h>
#include <mach/error.h>
#include <mach/mig_errors.h>
#include <mach/port.h>
#include <servers/netname.h>
#include <mach/mach_host.h>
#include <mach/host_info.h>
#include "cpu_server.h"
```

Este último fichero proviene de la generación de la interface con el servidor de procesadores por medio de MIG. Se encuentra en el directorio /usr/mach3/include.

Variables globales de comunicación entre la aplicación y el servidor:

```
mach_port_t             server_port;
mach_error_t            error;
processor_set_t         my_pset;
processor_set_name_t    pset_name;
int                     request,info,info_size;
```

Primero hay que buscar el puerto por el que nos comunicaremos con el servidor. Para ello nos ponemos en contacto con el servidor de nombres (**sname**s) y le preguntamos si tiene registrado algún servicio (*port*) con el nombre “UPC-cpu_server”:

```
error = netname_look_up(name_server_port,””,”UPC-cpu_server”, &server_port);
    if (error != NETNAME_SUCCESS) {
        mach_error(“netname_look_up”, error);
        exit(1);
    }
```

Tras ello, ya se pueden crear los *processor set*, para los cuales iremos posteriormente solicitando y liberando procesadores físicos:

```
error = processor_set_create(mach_host_self(),&my_pset,&pset_name);
if (error != KERN_SUCCESS) {
        mach_error(“processor_set_create”, error);
        exit(1);
    }
```

Uno de los objetivos de diseño de este servidor de procesadores es que las políticas de asignación de procesadores fueran sencillas. El servidor mantiene disponibles, para otorgar a cualquier task que le llama, hasta el número máximo de procesadores de la máquina menos uno, que es el procesador master y que es mejor no asignar en exclusiva a nadie (de hecho, no puede salir del *default processor set*).

El *server* no desbanca procesadores a las aplicaciones; son éstas las que los liberan en cuanto ya no los necesitan o acaban.

Si la task quiere que los threads que cree a partir de ahora vayan a parar a dicho *processor set*, debe asignarse ella misma:

```
error = task_assign(mach_task_self(),my_pset,0);
    if (error != KERN_SUCCESS)
        mach_error(“task_assign”, error);
```

El último parámetro (0) indica que los thread que ya tenía la task no sean transferidos al nuevo *processor set*. Por ahora es necesario que sea así, ya que no se ejecutará nada en el *processor set* hasta que tenga procesadores asignados. Cuando sea así, podremos mover, individualmente, los threads que nos resten por asignar con la llamada:

```
error = thread_assign(mach_thread_self(),my_pset);
    if (error != KERN_SUCCESS)
        mach_error(“thread_assign”, error);
```

Cuando la aplicación quiera acabar, tendría que devolver todos los procesadores al *default processor set* y destruir su propio *processor set*. No puede hacerlo de forma satisfactoria (continuando tras ello su ejecución) sin cambiar primero su thread al *default processor set*:

```
error = thread_assign_default(mach_thread_self());
    if (error != KERN_SUCCESS) {
        mach_error(“thread_assign_default”, error);
```

Ahora ya puede liberar los procesadores, como veremos más adelante, y destruir el *pro-*

cessor set:

```
error = processor_set_destroy(my_pset);
     if (error != KERN_SUCCESS) {
          mach_error("processor_set_destroy", error);
```

A.2.1. Cpu_server_request

La primera llamada del interface de nuestro servidor de procesadores es para solicitar N procesadores físicos, pidiendo que los asigne a uno de nuestros *processor set*:

```
request=N;
cpu_server_request (server_port, my_pset, &request, flavor);
```

Si el servidor dispone de suficientes procesadores libres, concede tantos como le pida la aplicación. Como resultado de la función devuelve KERN_SUCCESS si todo ha ido bien, y si no, o bien los errores que le envíe el servidor o bien los errores producidos por el mecanismo de mensajes. Si va bien, devuelve además en *reques*t el número real de procesadores asignados.

No quita nunca procesadores físicos asignados a otros *processor sets*, son las aplicaciones las que deben devolver los procesadores en cuanto no los utilicen.

Su funcionamiento depende del *flavo*r que le pase. Los valores que se aceptan son los siguientes:

- CPU_REQ_AMAP: Asigna tantos como puede. Devuelv*e* en *reques*t el número real asignado.
- CPU_REQ_ALL: O asigna todos los procesadores solicitados o no asigna ninguno y devuelve error.
- CPU_REQ_2N: Asigna un número de procesadores igual o inferior al solicitado, pero siempre potencia de 2. Devuelv*e* en *reques*t el número real asignado.
- CPU_REQ_WAIT: Modifica la opción CPU_REQ_ALL de forma que no vuelva hasta que realmente se le hayan asignado todos los procesadores.

A.2.2. Cpu_server_release

La segunda llamada del interface de nuestro servidor de procesadores es para liberar M procesadores físicos, pidiendo que los desasigne de uno de nuestros *processor set* y los asigne al *default processor set*:

```
request = M;
error = cpu_server_release (server_port, my_pset, &request);
```

Devuelve en *reques*t el número real de procesadores que ha podido liberar.

En esta primera versión no se puede indicar específicamente cuáles desearíamos liberar. En la próxima versión se pasará una lista de procesadores.

A.2.3. Cpu_server_info

La tercera llamada del interfaz de nuestro servidor de procesadores es para solicitar información acerca de los procesadores físicos gestionados por el servidor y asignados a la aplicación:

```
error = cpu_server_info (server_port, &info, flavor, &info_size);
```

Devuelve, en la variable info, si el flavor es cero (0), el número de procesadores actual-

mente libres, y si es uno (1), un vector de tamaño info_size con los identificadores (el *slot* en el bus) de los procesadores físicos asignados a la aplicación.

A.3. Detalles del diseño sobre Mach

A.3.1. Identificación de las peticiones

Mach es orientado a objetos. Para realizar una operación es necesario a) conocer el objeto y b) poseer derechos de realizar esa operación sobre el objeto. Ambas necesidades se concretan en el paradigma de trabajar con *capabilities*. Un *port* describe a cualquier objeto y la única manera de conocerlo es haberlo creado o haberlo recibido a través de un mensaje. No hay camino de vuelta, a partir de un puerto saber qué representa.

El CPU *server* sabe distinguir si una petición es para el mismo *processor set* que una anterior gracias al identificador del *processor set*. El cliente envía cada vez el mismo identificador de *processor set* (es un *port*, como todo descriptor de objeto), y el receptor recibe cada vez el mismo identificador (es un derecho de envío sobre el *port*, que permite realizar operaciones sobre el *processor set*). Es una peculiaridad del mecanismo de mensajes de Mach: los *ports* son del kernel y a las tasks les da sólo un identificador dentro de su propio espacio de direcciones (como un canal de UNIX). Los *ports* se pueden pasar de una task a otra insertándolos dentro de los mensajes. En este caso, el kernel reconoce que la task destino ya posee ese *port* y modifica el mensaje para que reciba el mismo identificador.

El CPU *server* mantiene el estado de los procesadores en una tabla. En ella relaciona los procesadores físicos con el pset al que están actualmente asignados y qué task solicitó la asignación. Los procesadores asignados al default pset, sin contar al master, están disponibles para ser reasignados.

A.3.2. Servidor multiflujo

Debido a que no todas la peticiones de servicio se pueden atender inmediatamente, el CPU *server* ha sido realizado con la posibilidad de creación dinámica de flujos que atiendan peticiones de reasignación de procesadores.

El usuario tiene la posibilidad de solicitar que no se le sirva su petición hasta que todos los procesadores que ha solicitado le hayan sido asignado. Es el flavor CPU_REQ_WAIT. Si el cthread que ha tomado esta petición se da cuenta de que no hay suficientes procesadores disponibles en ese instante, y que no quedaría ningún otro cthread a la espera de nuevas peticiones, crea un nuevo flujo para atenderlas y él se bloquea en una *condition* a la espera de procesadores.

Cada vez que se liberen procesadores, se despierta a los flujos que están esperando y estos reevalúan su petición. Si alguno consigue todos los procesadores que necesita, los asigna al pset y contesta al cliente. Si no hay suficientes, no asigna ninguno y vuelve a esperar.

A.3.3. Gestión de excepciones

El cliente debería liberar sus procesadores llamando a *cpu_server_releas*e antes de destruir su *processor set*. Con la destrucción también se consigue que los procesadores actualmente asignados a ese *processor set* el sistema los devuelva al *default processor set*, pero entonces el CPU *server* no se enteraría y se creería que los tiene aún asignados a la aplicación.

Mediante el mecanismo de atención de excepciones de Mach, el servidor solicita al kernel que le avise de los acontecimientos que afecten a los puertos de los psets y de las tasks que tiene registrados en su tabla de asignaciones. El kernel envía un mensaje al CPU *server* cuando el estado de uno de esos puertos cambia y así el servidor se entera de cuándo un *processor set* o una task a la

que había asignado procesadores, ha caído o ha sido destruído.

Un cthread del CPU *server* está dedicado al tratamiento de excepciones. Crea un puerto y comunica al kernel que ese será el puerto de excepciones del servidor. Se queda continuamente escuchando de ese puerto y, cuando le llegan mensajes de destrucción de algún objeto que conoce, actualiza la información del estado de los procesadores y psets.

B

Nuevo interfaz de la librería CThreads$^+$

"Chi vuoc por termine alli umani ingegni?"

Galileo Galilei

Apéndice B: Nuevo interfaz de la librería CThreads^{+}

B.1. Introducción 199

B.2. Planificación por prioridades 199

B.2.1. Modificar la prioridad: cthread_set_prio 199

B.2.2. Consultar la prioridad: cthread_get_prio 200

B.2.3. Inicializar el rango de prioridades: cthread_init_prio 200

B.3. Cesión voluntaria del procesador virtual 201

B.3.1. Cthread_handoff 201

B.3.2. Cesión del procesador virtual a otro flujo más prioritario: cthread_hint 202

B.1. Introducción

Se detalla en este apéndice el conjunto de funciones añadidas a la librería CThreads [COOP88] para la planificación de flujos dentro de la aplicación, a nivel usuario. Está formado por dos tipos de llamadas a la librería: las necesarias para dotar a la librería de prioridades a nivel usuario y las que ceden el procesador virtual -y con ello, el procesador físico- a otro cthread concreto o al de mayor prioridad. La nueva librería se denomina CThreads^{+}.

Las prioridades a nivel de usuario, nosotros las vemos más bien como un peso asociado al flujo que indica en qué momento, o en qué orden, respecto al resto de flujos de la aplicación, debe ser planificado.

Nuestro objetivo principal de diseño, en todo el entorno de ejecución realizado, ha sido siempre facilitar al usuario todas las herramientas necesarias de planificación, pero sin proporcionar nunca más semántica a cada operación de la que el propio usuario pide: si pide un cambio de flujo a nivel usuario y éste no es posible, devolverle el resultado al usuario y que sea él el que decida la nueva acción a seguir.

Para permitir el uso de prioridades en la librería de Cthreads, hemos añadido nuevas llamadas, en las que la aplicación expresa la prioridad mayor, la menor y la inicial con la que van a trabajar sus flujos. Son prioridades estáticas, que se adaptan mucho mejor al objetivo de uso para las que las queremos: que la aplicación controle la jerarquía de ejecución de sus flujos.

Inicialmente, la librería da por defecto un valor a las tres prioridades de la aplicación[1]: si el usuario no explicita unos valores diferentes, sus flujos se crean todos con la misma prioridad, que no se modifica, y se comportan de manera FIFO respecto a su inserción en las colas de preparados (el funcionamiento tradicional de Cthreads).

Dos nuevas llamadas permiten la cesión de procesadores virtuales (o threads de kernel) a nivel usuario.

Con la llamada cthread_hint*()*, un flujo replantea un cambio de contexto a la aplicación; este cambio será realmente efectivo si existe un flujo preparado para ejecutar con una prioridad mayor o igual al del actual. La diferencia básica con cthread_yield*()*, ya existente en la librería, es que nunca se intenta un cambio de contexto a nivel de kernel, sino sólo con flujos de usuario; si no es posible, se vuelve al primer flujo.

A través de la llamada cthread_handoff(thread), permitimos que un flujo de usuario dé paso directamente a otro, saltándose la política de planificación de la aplicación -en nuestro caso, sin comprobar la prioridad que tiene el flujo al que le pasamos control-.

B.2. Planificación por prioridades

La política de planificación de flujos de usuario de la nueva librería es por prioridades.Tres nuevas llamadas permiten inicializar, consultar y modificar las prioridad de un cthread.

B.2.1. Modificar la prioridad: cthread_set_prio

```
unsigned int cthread_set_prio (t, prio)
 cthread_t t;
 unsigned int prio;
```

1. Los valores que da la librería por defecto son 127 como prioridad mínima, 0 como máxima y 30 como prioridad por defecto. Como puede comprobarse, se ha seguido la política definida por UNIX para las prioridades máxima y mínima, por tradición.

Descripción

Comprueba que la nueva prioridad esté entre *cthread_max_prio* y *cthread_min_prio*.

Modifica la prioridad del cthread.

Si el thread no ha comenzado aún a ejecutarse, reordena la cola de cthreads. Si está ya corriendo (tiene cproc asociado), entonces modifica la prioridad de planificación y reordena la cola de preparados (*ready*).

Si la prioridad es mayor que la del cthread actual, llama a *cthread_hint()*.

Parámetros

cthread_t t	Puntero a la estructura cthread
unsigned int prio	Nueva prioridad

Valor que retorna

Devuelve la prioridad anterior

Si la prioridad no está dentro del rango permitido: error (-1)

B.2.2. Consultar la prioridad: cthread_get_prio

```
unsigned int cthread_get_prio(t)
 cthread_t t;
```

Descripción

Permite consultar la prioridad del cthread que se le pasa como parámetro.

Parámetros

cthread_t t	Puntero a la estructura cthread

Valor que retorna

Devuelve la prioridad actual del cthread

B.2.3. Inicializar el rango de prioridades: cthread_init_prio

```
void cthread_init_prio (minprio, max_prio, default_prio)
 unsigned int minprio, max_prio, default_prio;
```

Descripción

Inicializa (o reinicializa) los valores de las prioridades mínima, máxima y por defecto de los cthreads de esta task que se creen a partir de ahora o a los cuales se les modifique la prioridad. Modifica los valores de las variables globales:

cthread_max_prio (valor inicial: 127)

cthread_min_prio (valor inicial: 0)

cthread_default_prio (valor inicial: 30)

Parámetros

unsigned int minprio, max_prio, default_prio
valores decimales de las prioridades

Valor que retorna

Ninguno

B.3. Cesión voluntaria del procesador virtual

Dos nuevas rutinas para la planificación de cthreads a nivel de la librería. Su objetivo es la cesión del procesador virtual (thread de kernel) a otro cthread que se encuentre preparado para ejecutar en modo usuario. La primera cede el procesador virtual a un cthread determinado y la segunda lo cede en general al cthread que tenga la prioridad mayor.

B.3.1. Cthread_handoff

```
int cthread_handoff (t)
 cthread_t t;
```

Descripción

Intenta ceder el procesador virtual sobre el que está corriendo al cthread que pasa como parámetro (cthread destino).

Si el thread destino se encuentra en la cola de preparados (ready), lo extrae y provoca un cambio de contexto con ese cthread.

Comprueba que el que realiza la llamada no esté ligado a su procesador virtual (*wired*), que el thread destino exista y que esté listo para ejecución (tenga un cproc).

Comprueba que no esté ya corriendo en otro procesador virtual.

Si no está corriendo, es que está bloqueado.

Parámetros

cthread_t t   Puntero a la estructura cthread

Valor que retorna

0: Ha conseguido ceder correctamente el procesador virtual. Devuelve este valor cuando el cthread es vuelto a seleccionar para ejecución.

1: Error, el puntero a la estructura cthread del cthread destino es inválido.

2: Error, el cthread destino no tiene aún todo su entorno preparado para ejecutar (no tiene cproc asociado). No es aún candidato a ejecutar.

3: Error, el cthread destino está ya corriendo en otro procesador.

4: Error, el cthread destino está bloqueado, no puede ejecutarse.

5: Error, el cthread actual está ligado a su procesador virtual y no puede cederlo.

Información adicional

Si estamos trabajando en el entorno de “eXecution contexts”, actualiza los valores de

quién está corriendo en este procesador virtual.

B.3.2. Cesión del procesador virtual a otro flujo más prioritario: cthread_hint

```
int cthread_hint()
```

Descripción

Intenta ceder el procesador virtual sobre el que corre a otro cthread preparado para ejecutar, según la política de prioridades de la nueva versión de la librería.

Comprueba que el cthread que seleccionaría tiene una prioridad mejor que la del cthread actual. Si no, devuelve el control.

Extrae de la cola de preparados al nuevo cthread e inserta el actual.

Comprueba que el cthread actual no esté ligado al procesador virtual.

Actualiza el estado de los dos cthreads.

Provoca un cambio de contexto con ese cthread.

Si no encuentra un cthread en la cola de preparados, devuelve el control.

Parámetros

Ninguno

Valor que retorna

Si no existe un cthread preparado más prioritario, devuelve el control (con valor 0).

Si no encuentra cthread en preparados, devuelve error (1).

Si el cthread actual está ligado a su procesador virtual, devuelve error (5).

Información adicional

Si estamos trabajando en el entorno de "eXecution contexts", actualiza los valores de quién está corriendo en este procesador virtual.

Se diferencia de la llamada cthread_yield() de la librería tradicional en el hecho de que retorna si no encuentra un cthread preparado, en cambio cthread_yield() puede llamar al kernel y ceder el procesador virtual a otro thread cualquiera.

C

Funciones e interfaz del mecanismo de upcalls

"Y en la altura se vive
sin sentir la fatiga
de haber subido."

Pedro Salinas

Apéndice C: Funciones e interfaz del mecanismo de upcalls

C.1. Introducción ... 205

C.2. Inicialización del interfaz del mecanismo de upcalls ... 205
- C.2.1. Inicialización del entorno de planificación de flujos sobre eXc ... 205
- C.2.2. Registro de las rutinas de upcall ... 205
- C.2.3. Creación del espacio para el pool de pilas ... 206
- C.2.4. Solicitud de la activación del mecanismo de upcalls ... 207
- C.2.5. Programación de un temporizador ... 207

C.3. Interface de las upcalls (rutinas que atienden a cada upcall) ... 208

C.4. Interface interno a las rutinas de tratamiento de las upcalls ... 209
- C.4.1. Salvar el contexto de usuario de un eXc ... 209
- C.4.2. Encolar en la aplicación un eXc bloqueado en el kernel ... 210
- C.4.3. Desencolar en la aplicación un eXc desbloqueado en el kernel ... 210
- C.4.4. Inserción de un cthread en la tabla de preparados ... 210
- C.4.5. Cesión del eXc a un flujo planificador ... 210
- C.4.6. Planificación de un Cthread desde una upcall ... 211
- C.4.7. Continuación explícita de un Cthread desbancado ... 211

C.5. Flujos nulos y planificadores a nivel de aplicación ... 211
- C.5.1. Handoff de un cthread al flujo nulo ... 212
- C.5.2. Actualización de la cola de preparados ... 212
- C.5.3. Intento de bloqueo de un flujo nulo y cesión de su eXc ... 212
- C.5.4. Nueva primitiva de spin_lock: marcar quién está en posesión del lock ... 212
- C.5.5. Inhibición temporal del mecanismo de upcalls ... 213

C.1. Introducción

Este apéndice presenta el interfaz de llamadas desde la aplicación al kernel y del kernel a la aplicación que permite la cooperación en la planificación de sus flujos, tal como se ha visto en los capítulos 5 y 6.

Presentamos primero la inicialización del entorno de "eXecution contexts" (eXc) por parte de la aplicación. En segundo lugar, el formato de las llamadas y los parámetros que sube el kernel hacia la aplicación cuando se produce una upcall. Después, las rutinas internas a la programación de las upcalls, disponibles para aquellos programadores que quieran hacerse su propio tratamiento de las notificaciones de los eventos que ocurren en el kernel. En último lugar, se muestra el tratamiento por defecto que proponemos para usuarios no expertos y el funcionamiento de los flujos nulos y planificadores a nivel de usuario.

Estas funciones han sido añadidas a la librería CThreads$^+$, que ha pasado a llamarse libcthreads_exc.a para mantener la anterior intacta.

C.2. Inicialización del interfaz del mecanismo de upcalls

Los ficheros que llamen a esta interfaz deben incluir:

```
#include <cthreads_exc.h>
```

C.2.1. Inicialización del entorno de planificación de flujos sobre eXc

En primer lugar, hay que inicializar las tablas de mapeo entre contextos de ejecución (eXc), flujos de usuario y procesadores físicos. También hay que inicializar un flujo nulo por procesador y evitar que la creación de cthreads llegue a crear threads de kernel. Todo ello lo realiza la función eXc_init().

```
eXc_init();
```

Descripción

Inicializa la tabla proc_map, que relaciona flujo, eXc, procesador y flujo nulo.

Limita la creación de nuevos threads de kernel no asociados a un procesador, cthread_set_kernel_limit (1).

Crea e inicializa un flujo nulo planificador por cada procesador, llamando a sched_thread_init (processor)

C.2.2. Registro de las rutinas de upcall

El usuario registra los procedimientos que se activarán en las upcalls con la llamada:

```
kern_return_t eXc_upcall_check_in(upcall_id, routine)
int upcall_id;
vm_offset_t *routine;
```

Descripción

- Registra una rutina como un punto de entrada para el tratamiento de un evento en el espacio de direcciones de la task que hace la llamada.

El parámetro “upcall_id” es el número de la upcall que se quiere programar; está asociado a un evento. “routine” es una variable que contiene la dirección de la rutina del espacio de usuario, donde empezará a ejecutarse la upcall.

Las upcalls están asociadas a eventos que ocurren en el kernel. Actualmente están definidos los eventos:

```
#define NULL_EVENT 0
#define PROCESSOR_ADDED 1
#define PROCESSOR_PREEMPTED 2
#define BLOCKED_THREAD 3
#define RESUMED_THREAD 4
#define TIMER 5
```

Comprueba que “upcall_id” sea uno de estos eventos y que la dirección de la rutina sea del espacio de direcciones válido de la task.

Retorna en “routine” la rutina, si la hay, que anteriormente hubiera registrada para ese evento.

Retorna un error != KERN_SUCCESS si ha habido problemas.

Es una llamada directa al kernel de Mach (un trap).

Ejemplo:

```
void (*f1)();
f1 = upcall_processor_added;
error = upcall_check_in (PROCESSOR_ADDED, &f1);
```

C.2.3. Creación del espacio para el pool de pilas

Es necesaria la creación de un espacio de memoria en modo usuario para las pilas que utilizarán las upcalls:

```
int eXc_stack_init (n_stacks, stack_size)
int n_stacks, stack_size;
```

Descripción

El parámetro “n_stacks” indica el número de pilas que deseamos reservar, cada una de ellas de tamaño “stack_size”, en bytes.

Este pool de pilas es un recurso de la task. La llamada reserva el espacio de memoria necesario para albergar esas pilas.

El tamaño de las pilas es ajustado a múltiplo de página.

Retorna error si no se ha conseguido reservar el espacio pedido.

Rellena una estructura global de la aplicación, llamada eXc_stacks, compuesta, para cada pila, por un indicador de pila libre y un puntero al inicio de la pila.

```
typedef struct stack_mem {
 int busy;
 vm_offset_t stack_p;
} stack_mem_t;
stack_mem_t eXc_stacks[n_stacks];
```

Es una función interna de la librería libcthreads_exc.a.

Ejemplo:

```
eXc_stack_init (5, 2000);
```

C.2.4. Solicitud de la activación del mecanismo de upcalls

Utilizando esta llamada, la task actual decide activar o desactivar el mecanismo de upcalls y el funcionamiento de los eXc (execution context) en el grupo de procesadores al que está asignada. Informa al kernel de la activación del mecanismo a la vez que proporciona la dirección de la estructura que apunta al pool de pilas preparadas para las upcalls. Se recomienda llamarla cuando se haya creado un nuevo *processor set* y la aplicación se haya asignado a él. También conviene que se haga desde el primer flujo de usuario. El kernel devuelve el identificador del eXc asignado al flujo que ha hecho la llamada y el procesador sobre el que corre.

```
kern_return_t eXc_enable (enable, stack_pool, current_exc, current_processor)
boolean_t enable;
vm_offset_t stack_pool;
unsigned int *current_exc, *current_processor;
```

Descripción

Si enable>0, activa el mecanismo de *scheduler activations* para la aplicación. Si enable=0, lo desactiva.

Se le pasa la dirección inicial de la estructura que describe el pool de pilas para las upcalls. Comprueba que las direcciones son del espacio de usuario.

Inicializa los campos de la task que activan los eXc y las pilas.

Devuelve KERN_SUCCESS si todo va bien.

Devuelve el valor del identificador del eXc y el procesador actual, para inicializar las estructuras que permitirán la planificación de flujos a nivel usuario.

C.2.5. Programación de un temporizador

Programación para que se dispare la upcall de temporización tras el paso de un cierto intervalo de tiempo.

```
kern_return_t eXc_timer_set(time, flavor)
int time;
int flavor;
```

Descripción

Programación de una temporización para la aplicación, que disparará una upcall de TIMER cuando expire.

“time” indica el tiempo en centésimas de segundo.

“flavor” permite programar el *timer* para que se dispare una sola vez (flavor=0), o que se dispare repetidamente cada “time” centésimas de segundo, hasta que se desprograme.

Time = 0 desprograma una temporización con repetición.

Comprueba que la aplicación ha activado el funcionamiento por eXc, y que la upcall del

TIMER ha sido inicializada. Si no es así, devuelve KERN_INVALID_ARGUMENT. Si todo es correcto, devuelve KERN_SUCCESS.

Permite programar varios *timeouts* por aplicación, incluso que se solapen. Limitaciones: Solo se acuerda de la ultima programación con repetición; puede agotar el número de temporizaciones pendientes dentro del kernel (actualmente lo hemos inicializado a treinta, pero las compartimos con algunos dispositivos).

C.3. Interface de las upcalls (rutinas que atienden a cada upcall)

Cuando en el kernel ocurre un evento que hay que notificar al usuario, se activa el mecanismo de preparación y subida de una upcall. Dependiendo del evento, la información que proporciona el kernel al usuario varía. En nuestra realización, siempre se sube el tipo de evento acontecido, el procesador que está tratando el evento y el identificador de eXc que está subiendo la información. Además:

- Cuando se añade un procesador a la aplicación, se informa del procesador físico que se le ha asignado, para que la aplicación pueda actualizar su información respecto a los recursos que el sistema le confiere.
- Cuando un flujo se bloquea, se sube la identificación del eXc en que corría el flujo que se ha bloqueado. En el eXc que sube la notificación, la aplicación puede planificar a otro de sus flujos.
- Cuando un flujo se desbloquea, se sube el identificador del eXc que se ha desbloqueado, con todo su contexto modificado por la acción del kernel que provocó el bloqueo. Además, como ha habido que desbancar a un flujo asíncronamente, todo el contexto necesario del flujo desbancado, por si la aplicación no lo selecciona en ese instante para seguir ejecutándose.
- Cuando llega un temporizador, el contexto del flujo que estaba corriendo en ese procesador, por si acaso la aplicación decidiera un cambio de contexto.
- Cuando se quita un procesador a la aplicación, el procesador que se le ha quitado, el eXc que estaba corriendo en dicho procesador para que la aplicación pueda identificar el flujo y también actualizar su información, y el contexto de dicho flujo. Como para subir la notificación se ha tenido que desbancar a otro procesador de la aplicación, también se sube la información correspondiente: el procesador desbancado, para identificar el eXc y el flujo, y el contexto.

Podemos ver a continuación un ejemplo de la especificación de los parámetros que recibe cada upcall. Son las cabeceras de las rutinas por defecto que ofrece la librería CThreads^{+}. El parámetro “chain” apunta hacia la información de los otros eXc que se suben junto al que aporta la notificación.

```
void upcall_processor_added(event,new_sa,processor,stack,chain)
unsigned int event;
unsigned int new_sa;
unsigned int processor;
unsigned int stack;
unsigned int *chain;

void upcall_blocked_thread(event,new_sa,processor,stack,old_sa,chain)
unsigned int event;
unsigned int new_sa,old_sa;
unsigned int processor;
unsigned int stack;
unsigned int *chain;
```

```
void
upcall_processor_preempted(event,new_sa,processor,stack,thread_registers,chai
n)
unsigned int event;
unsigned int new_sa;
unsigned int processor;
struct i386_thread_state thread_registers;
unsigned int stack;
unsigned int *chain;

void upcall_timer(event,new_sa,processor,stack,old_sa,thread_registers,chain)
unsigned int event;
unsigned int new_sa,old_sa;
unsigned int processor;
struct i386_thread_state thread_registers;
unsigned int stack;
unsigned int *chain;

void
upcall_resumed_thread(event,new_sa,processor,stack,thread_registers,chain)
unsigned int event;
unsigned int new_sa;
unsigned int processor;
struct i386_thread_state thread_registers;
unsigned int stack;
unsigned int *chain;
```

C.4. Interface interno a las rutinas de tratamiento de las upcalls

Estas funciones se deben usar desde las rutinas de atención a upcalls para salvar los estados de los flujos que nos llegan desde el kernel y para dar control al flujo planificador de nuestro procesador. Hay que evitar interactuar con la librería CThreads, ya que la upcall no es un flujo reconocido por la librería. No puede bloquearse ni caer en un *spin lock*, pues durante la ejecución de la upcall están inhibidas otras upcalls en el mismo procesador.

C.4.1. Salvar el contexto de usuario de un eXc

Cuando entre los parámetros que suben en una upcall encontramos los registros de un flujo de usuario, salvados al entrar en el kernel, debemos copiarlos ordenadamente sobre la pila del flujo, de forma que después se pueda restaurar su ejecución desde las rutinas de planificación de la librería de cthreads.

```
unsigned int * eXc_move_stack(sa_sp,ct_sp)
struct i386_thread_state *sa_sp;
unsigned int *ct_sp;
```

Descripción

Copia desde el contexto del eXc que ha subido del kernel como parámetro de la upcall, a la pila del cthread que ha sido desbancado en modo usuario o se ha desbloqueado en el kernel. Copia los valores de los registros de forma que los encuentren como esperan cthread_switch() o cproc_switch(). Además, empila la continuación explícita a sa_switch(), que restaurará el resto de registros.

Devuelve el nuevo valor del puntero a la pila del cthread (para almacenarlo en el contexto del cthread).

C.4.2. Encolar en la aplicación un eXc bloqueado en el kernel

```
void eXc_blocked_enqueue(th,exc_id)
cproc_t th;
unsigned int exc_id;
```

Descripción

Inserta un cthread en la cola a nivel usuario de flujos bloqueados dentro del kernel. La cola es nueva y se llama eXc_blocked. Se utiliza para guardar los cthreads que nos han notificado que se han quedado dentro del kernel.

Modifica el estado del cthread apuntado por th a CPROC_BLOCKED.

C.4.3. Desencolar en la aplicación un eXc desbloqueado en el kernel

```
cproc_t eXc_blocked_dequeue(exc_id)
unsigned int exc_id;
```

Descripción

Recorre la cola a nivel usuario de flujos bloqueados dentro del kernel (eXc_blocked) hasta encontrar el identificado por exc_id (suele ser el identificador del contexto de ejecución que se bloqueó y que ahora nos han notificado que ya se ha desbloqueado.

Si lo encuentra, devuelve el puntero al cthread que ha desencolado.

Si no lo encuentra, devuelve NO_CPROC.

C.4.4. Inserción de un cthread en la tabla de preparados

Los descriptores de los flujos que suban con los parámetros de la upcall y que se encuentren en estado RUNNING, ya sea por haber sido desbancados por la notificación, o porque acaban de desbloquearse, deben insertarse en la cola de preparados de la librería CThreads. Como esa operación requiere una exclusión mutua que no conviene desde un tratamiento de upcall, se insertan en una extensión de esa cola que no requiere ninguna sección crítica por ser local al procesador la operación de insertar.

```
eXc_ready_queue_insert(th, processor)
cproc_t th;
unsigned int processor;
```

Descripción

Inserta el puntero al descriptor al cthread th en la parte de la cola de threads preparados para ejecutar correspondiente al procesador processor.

La estructura donde se almacena debe tener una dimensión suficiente para poder almacenar los descriptores sin problemas de espacio. Actualmente es una tabla (ready_table) de tamaño MAX_CPU * MAX_EXC.

C.4.5. Cesión del eXc a un flujo planificador

En cuanto la rutina de tratamiento de la upcall haya salvado el estado de los flujos implicados, insertado su descriptor en la cola de bloqueados o en la tabla de preparados, y haya actualizado el mapeo procesador-eXc, debe ceder el control a un flujo planificador. Con ello se consigue liberar la inhibición de nuevas upcalls y que sobre el eXc en curso se ejecute un cthread reconocido por la librería y que por tanto puede entrar en exclusión mutua con otros, planificar, etc...

```
eXc_sched_thread_handoff(processor, stack)
unsigned int processor, stack;
```

Descripción

Localiza el flujo planificador del procesador donde ha ocurrido la upcall.

Lo planifica sobre el eXc en curso, llamando a cproc_dispatch(&planificador->sp, stack).

No retorna, a no ser que el flujo planificador ya estuviera corriendo, entonces es un error (1).

C.4.6. Planificación de un Cthread desde una upcall

Llamada desde una upcall, una vez seleccionado ya un cthread para correr sobre el nuevo contexto de ejecución. Es un cambio de flujo a nivel usuario, pero no hay estado a salvar, pues venimos de una upcall. Se pasa también el identificador de la pila que está usando la upcall para que la libere y devuelva al pool de pilas reciclables por las upcalls.

```
void eXc_cproc_dispatch (context, discard_stack)
unsigned int *context;
unsigned int discard_stack;
```

Descripción

Es código ensamblador dependiente de la máquina.

Cambia de pila, salta a la pila del nuevo cthread (apuntada por el parámetro context).

Marca la pila de la upcall como FREE, para que se pueda reutilizar, desinhibiendo la subida de nuevas upcalls.

Restaura algunos registros del cthread y vuelve al código del cthread que provocó el bloqueo síncrono en modo usuario o a la continuación explícita que estableció eXc_move_stack() para un flujo desbancado por una upcall.

C.4.7. Continuación explícita de un Cthread desbancado

```
void eXc_switch()
```

Descripción

Es la continuación de usuario para cthreads desbancados. Suele venir directamente de eXc_cproc_dispatch(), por continuación explícita.

Es código ensamblador dependiente de la máquina.

Restaura algunos registros del cthread que no se suelen salvar en un bloqueo síncrono.

Vuelve al código del cthread desbancado en modo usuario.

C.5. Flujos nulos y planificadores a nivel de aplicación

Hemos dotado a la librería CThreads$^+$ de unos flujos que realizan la función de flujos nulos en los momentos en que no hay flujos preparados para correr sobre los procesadores que nos ofrece el kernel, y también cumplen la función de seleccionar el siguiente flujo a ejecutar, según la política de la librería. Hay uno por procesador asignado a la aplicación y se le da control por *handoff* explícito sobre él mediante la llamada eXc_sched_thread_handoff(processor, stack) desde una upcall y la llamada eXc_handoff_sched_thread(th, processor) desde un cthread.

Las primitivas cthread_yield(), cthread_hint() y cthread_handoff(th), explicadas en el apéndice anterior, realizan el cambio de contexto entre flujos de usuario sin pasar por el flujo nulo, pero para ello han sido reescritas de forma que actualicen la cola de flujos preparados y seleccionen el siguiente flujo como veremos que hace el flujo planificador.

C.5.1. *Handoff* de un cthread al flujo nulo

Cuando la librería o la aplicación detecta que en ese instante no hay trabajo útil para planificar sobre un eXc, y el flujo actual se acaba de bloquear, cede el control al flujo nulo del procesador mediante la primitiva:

```
eXc_handoff_sched_thread (th, processor)
cproc_t th;
int processor;
```

Descripción

Selecciona al cthread que es el flujo nulo del procesador "processor"

Actualiza la información de planificación de usuario, el mapeo procesador-eXc-flujo

Cambia de contexto llamando a cproc_switch().

C.5.2. Actualización de la cola de preparados

Las rutinas de atención a upcalls insertan, como hemos dicho ya, los flujos en estado preparado en una extensión de la cola de preparados local a cada procesador que no necesita exclusión mutua. Por ello, todos los que quieran seleccionar un nuevo cthread, tendrán primero que unificar la información en una sola cola. Esa operación la hace el primero que coge la exclusión mutua sobre la cola, operación obligatoria para extraer un elemento.

```
eXc_ready_table_to_ready_queue ();
```

Descripción

Debe tenerse el l*ock* ready_lock cogido antes de llamar a esta rutina.

Mueve los cthreads preparados de las colas de los procesadores a la cola central de la librería.

C.5.3. Intento de bloqueo de un flujo nulo y cesión de su eXc

El bucle principal de un flujo planificador a nivel usuario será actualizar la cola de flujos preparados y a continuación intentar ceder su eXc a un cthread preparado y quedarse él bloqueado en una estructura local al procesador, en modo usuario.

```
eXc_sched_thread_suspend ();
```

Descripción

Trata de desencolar un cthread preparado de la cola de flujos preparados y si lo halla, le cede el control llamando a cproc_switch().

Si no hay nadie preparado, retorna un 1.

C.5.4. Nueva primitiva de *spin lock*: marcar quién está en posesión del *lock*

La pareja de funciones eXc_spin_try_lock() y eXc_spin_unlock() permiten un acceso a la

cola de flujos preparados de la librería CThreads+ de forma segura, evitando los problemas que surgen al permitir el desbanque de flujos por las upcalls, y gracias a ello, el abrazo mortal.

```
boolean eXc_spin_try_lock (int * who_has_lock, int who_am_i);

eXc_spin_unlock(who_has_lock);
```

```
boolean eXc_spin_try_lock (int * who_has_lock, int who_am_i)
{

   while (spin_lock_locked (&ready_lock) ||
          eXc_no_preemption ()       ||
          (!spin_try_lock (&ready_lock)  && eXc_preemption ())) ;
   if (*who_has_lock!=0) {
      spin_unlock (&ready_lock);
      eXc_preemption ();
      return 0;
   }
   *who_has_lock = who_am_i;
   spin_unlock (&ready_lock);
   eXc_preemption ();
   return 1;
}

#define eXc_spin_unlock(who_has_lock) (*(who_has_lock)) = 0
```

Figura C-1 Esquema del código de la nueva primitiva de *spin lock* para acceder a la cola de preparados.

Descripción

Esta es la rutina (Figura C-1) que permite intentar coger el *lock* de la cola de preparados y si lo consigue, a la vez actualiza la variable global de la librería que indica quién posee en este instante este *lock* tan crucial para el cambio de contexto a nivel usuario. Si otro cthread tiene la exclusión mutua reservada, retorna FALSE, si lo ha conseguido, TRUE.

Para evitar una actualización incorrecta si es desbancada por una upcall, inhibe el mecanismo de subida de upcalls durante el corto espacio de tiempo de consulta y actualización de quién posee el *lock* de la ready *queue*. Para lo cual utiliza las dos rutinas siguientes.

C.5.5. Inhibición temporal del mecanismo de upcalls

La aplicación puede inhibir durante un corto espacio de tiempo la aparición de flujos de tratamiento de upcalls en su espacio de direcciones. Es necesario para evitar los problemas que pudieran derivarse.

```
eXc_no_preemption ();

eXc_preemption ();
```

Descripción

eXc_no_preemption() modifica un flag compartido entre el kernel y la aplicación de forma que el primero retrasa la notificación de eventos que sucedan en el kernel hasta que se vuelva a permitir llamando a eXc_preemption().