Mach

de pe a pa

José E. Marchesi (jemarch@gnu.org)

Mach de pe a pa, versión 1.0.

Copyright © 2003 José E. Marchesi.

Se permite la copia, distribución y la modificación de este documento bajo los términos de la GNU Free Documentation License, Version 1.1 o cualquier versión posterior publicada por la Free Software Foundation; sin secciones invariantes, siendo el texto de portada "Mach de pe a pa", siendo el texto de la contraportada el especificado en (a), mas abajo. Se incluye una copia de la licencia en la sección titulada "Free Documentation License".

(a) El texto de la contraportada de "Mach de pe a pa" es: "Dispones de libertad para copiar y modificar este libro."

Índice General

1 Prefacio

2 Introducción

Este documento existe por una razón (como prácticamente todos los documentos, exceptuando probablemente los manuales de visual basic).

Yo soy, principalmente, un desarrollador de UNIX. Debido a ello, tengo mi mente bastante POSIXficada. Cuando comencé a estudiar algo de mach, esto resultó ser en ocasiones un problema. Mach se compone de pocos conceptos, pero esos conceptos son radicalmente nuevos en comparación con los conceptos POSIX de toda la vida. Por eso pueden resultar difíciles de comprender al principio.

Este documento existe para que los duros de mollera como yo tengan una ayudita al asimilar esos conceptos.

La cosa no está tan chunga, ¿sabéis?. Si alguien como yo (mas furro que un pedazo de madera) está ahora manteniendo su propia versión de mach (llamada gnu-osfmk) para powerpc (bueno, en realidad para mi ibook), entonces vosotros, seres inteligentes, podréis dominar el mundo.

En fin, que espero que esto os sirva de ayuda.

Por supuesto, mis disculpas de antemano ante cualquier error que pudiera contener este documento.

3 Arquitectura de Mach

3.1 Hilos y Tareas

3.1.1 Hilos

(NOTA: Algunas cosas de este apartado tal vez no estén muy claras hasta que se lea el siguiente: "Tareas". Paciencia :)

Los hilos son las entidades computacionales básicas en Mach. Esto quiere decir que, en Mach, las "cosas" que ejecutan instrucciones son los hilos. Como veremos, hay tendencia a confundir un hilo con la tarea que lo contiene. Pese a que existe una relación muy estrecha entre un hilo y su tarea contenedora, esta confusión debe erradicarse de raíz cuanto antes. La entidad ejecutora es el hilo, no la tarea.

Un hilo pertenece a una (y solo una) tarea, que define su espacio de direcciones virtuales.

Los hilos son lo mas ligeros posible, consistiendo únicamente en un estado de procesador: conjunto de registros del procesador. Por ello, los cambios de contexto entre hilos (interrumpir la ejecución de uno y "dar el procesador" a otro) son poco costosos.

Las únicas acciones que puede llevar a cabo un hilo de forma autónoma es modificar el valor de los registros del procesador y leer y escribir en el rango de memoria virtual definido por la tarea que lo contiene. Para cualquier otra cosa, el hilo debe ejecutar trap especiales, que provocan que el kernel realice operaciones en nombre del hilo, o que el kernel envíe un mensaje a otra entidad reclamando la operación, también en nombre del hilo. Esto quedará claro en los apartados siguientes.

3.1.1.1 Estructura de Cliente visible de un hilo

Para invocar operaciones sobre un hilo se utiliza el puerto de kernel del hilo. Este puerto es utilizado, principalmente, por la tarea que contiene el hilo, por el conjunto de procesadores (processor set) en el que se ejecuta el hilo, y por otras tareas que tengan derechos de escritura en el puerto.

Evidentemente, el derecho de leer del puerto de kernel del hilo lo posee el propio hilo. Ese derecho no es transferible.

El puerto de kernel de un hilo se obtiene cuando se crea el hilo.

Además, el hilo posee un derecho de escribir para el puerto de excepciones del kernel del hilo. El destinatario de ese puerto es el kernel.

3.1.2 ¿Qué puede hacer un hilo?

Un hilo puede efectuar acciones de dos formas:

Directamente, mediante traps especiales

Un trap es una interrupción software. Parte de ellas las atiende el kernel (mach), y de ese modo ofrece servicios.

Otras no están definidas.

Indirectamente

Ganando un derecho de escritura en un puerto, y enviando mensajes a ese puerto. El destinatario del puerto será una tarea (o el propio kernel) que lleve a cabo la acción deseada.

En cuanto a enviar mensajes, las operaciones dependen del objeto que recibe los mensajes (que es manipulado).

Veamos el conjunto de traps que puede invocar un hilo.

3.1.2.1 Traps de scheduling

Estos traps afectan a la planificación del hilo por parte del conjunto de procesadores que lo contienen.

`thread_switch`
: Provoca un cambio de contexto con varias opciones.

  Mach permite que un hilo pida un cambio de contexto para realizar bloqueos por software (implementar una región crítica, un semáforo, un monitor, u otra primitiva de concurrencia).

`evc_wait`
: Provoca que el hilo espere suspendido hasta que se de un evento definido en el kernel (de un dispositivo).

3.1.2.2 Traps de identificación

Ya hemos comentado que, aparte de unos pocos traps, un hilo debe enviar mensajes para llevar a cabo acciones. Esto incluye operaciones sobre sí mismo (como hilo) o sobre la tarea que lo contiene.

Para ello, evidentemente, el hilo necesita un derecho de escribir en su puerto de kernel, y otro derecho de escribir en el puerto de kernel de la tarea. La obtención de estos puertos es un proceso conocido como bootstrap, y se utilizan los dos traps siguientes.

`mach_thread_self`
: Se retorna un derecho de escribir del puerto de kernel del hilo.

`mach_task_self`
: Se retorna un derecho de escribir del puerto de kernel de la tarea que contiene al hilo.

`mach_host_self`
: Se retorna un derecho de escribir del puerto de kernel del host en que se ejecuta el hilo.

`mach_reply_port`
: Se crea y retorna un derecho de recibir para un puerto. Este puerto se utiliza para recibir respuestas a mensajes que requieran una.

3.1.2.3 Trap de envío de mensajes

`mach_msg_trap`
: Este trap es el mas importante, ya que permite la utilización de mensajes por parte del hilo. Este trap es invocado por la llamada de biblioteca 'mach_msg'.

  Se encarga de enviar/recibir un mensaje a/desde un puerto identificado por un derecho pasado como argumento.

  La semántica de este trap es muy elaborada y compleja. Los detalles se ocultan al programador mediante el uso de MIG, el generador de interfaces.

3.1.2.4 Tratamiento de excepciones

Un hilo tiene un puerto de excepciones asociado. Cuando ocurre una excepción, el hilo envía un mensaje al puerto de excepciones, definiendo la excepción.

Una respuesta "succesful" a este mensaje hace que el hilo siga ejecutándose. En caso contrario, el hilo se temina.

Si el puerto de excepciones del hilo no existe, entonces el kernel intenta enviar el mensaje de excepción al puerto de excepciones de la tarea que contiene al hilo. Si no existe el puerto de excepciones de la tarea, el hilo se termina.

Hay dos excepciones :) a este mecanismo.

- Una falta de página no provoca un mensaje al puerto de excepciones. Lo que ocurre es que se envía un mensaje al gestor de memoria externo asociado a la página donde ocurrió la falta.
- El mecanismo general de excepciones no se aplica en el caso de las llamadas al sistema. Inicialmente, algunas de las llamadas al sistema las utiliza el kernel, y el resto están sin definir. Cualquier intento de invocar a una llamada al sistema no definida resulta en una excepción.

  Sin embargo, en base a cada tarea, es posible fijar una indirección asociada a un número de llamada al sistema. Esto se hace mediante task->task_set_emulation o `task->task_set_emulation_vector` (y examinado con `task->task_get_emulation_vector`).

  Este mecanismo permite que una tarea fije tratamientos a llamadas al sistema, en forma de rutinas. Es un buen modo de emular las llamadas al sistema de los sistemas operativos que se emulen sobre mach.

3.1.3 Acciones sobre hilos

Las siguientes acciones se efectúan enviando mensajes adecuados al puerto de kernel del hilo. Por tanto, es necesario poseer un derecho de escritura en dicho puerto.

3.1.3.1 Creación y destrucción de hilos

Una tarea puede crear hilos dentro de ella mediante

```
task->thread_create
```

obsérvese que este mensaje se envía al puerto de kernel de la tarea.

Puede destruirse un hilo enviándose un mensaje al puerto de kernel del hilo:

```
thread->thread_teminate
```

El resultado de task->thread-create es la obtención de un derecho de escritura en el puerto de kernel del hilo recién creado.

Puede pedirse a una tarea una lista con los derechos de escritura en los puertos de kernel de todos los hilos que tenga mediante el mensaje:

```
task->task_threads
```

Cuando se crea un hilo, se lo pone en estado 'suspendido'. Es igual que si se hubiera enviado el mensaje `thread->thread_suspend` al puerto de kernel del hilo justo antes de que el hilo ejecutara la primera instrucción. Para sacar al hilo de su estado suspendido (de hecho, para decrementar su contador de suspensión) se utiliza el mensaje `thread->thread_resume`

3.1.4 Estado de un hilo

El estado de un hilo viene definido por dos conjuntos:

- El estado de procesador (el valor de los registros).
- Un conjunto de puertos especiales.

3.1.4.1 Estado de procesador

El estado de procesador de un hilo se obtiene mediante

```
thread->thread_get_state
```

y se fija mediante

```
thread->thread_set_state
```

Es necesario un protocolo para cambiar el estado de procesador de un hilo, sin obtener resultados aleatorios (y desastrosos!):

1. Se suspende el hilo: `thread->thread_suspend`

   Esto no es necesario si el hilo se acaba de crear, ya que lo hace en estado suspendido.
2. Se aborta el hilo: `thread->thread_abort`

   Esto provoca que cualquier llamada al sistema que estuviera ejecutando el hilo temine inmediatamente. Si esa llamada al sistema era 'mach_msg' (estaba enviando o recibiendo un mensaje) se fija el contador de programa justo después de la llamada, con el resultado de "error de envío".

   Si se estaba efectuando una excepción o una falta de página, estas se reanudan en cuanto el proceso se despierte.
3. Se fija el nuevo estado: `thread->thread_set_state`
4. Se despierta al proceso: `thread->thread_resume`

3.1.4.2 Puertos especiales de un hilo

Un hilo posee dos puertos especiales asociados:

- El que utiliza para operaciones sobre sí mismo (*self*).
- El segundo es el puerto de excepciones.

El primer puerto suele ser el mismo que el puerto de kernel del hilo, si bien puede ser distinto (habitualmente fijado por el creador del hilo. Ver nota al final del documento).

Estos dos puertos pueden recogerse con

```
thread->thread_get_special_port
```

y fijados mediante

```
thread->thread_set_special_port
```

El resto del estado del hilo (el contador de suspensión y la información de planificación) pueden obtenerse mediante

```
thread->thread_info
```

3.1.5 Control de planificación

Los siguientes mensajes afectan el modo en que el hilo destinatario es planificado por el kernel para ejecución (en qué cola, con qué prioridad, con qué política, etc). Los veremos en detalle cuando desarrollemos los conjuntos de procesadores (processor sets).

- `thread->thread_assign`
- `thread->thread_assign_default`
- `thread->thread_get_assigment`
- `thread->thread_max_priority`
- `thread->thread_policy`
- `thread->thread_priority`
- `host_control(thread)->thread_wire`

3.2 Tareas

Una tarea contiene:

- Un conjunto de hilos.
- Un espacio de nombres de puertos.
- Un espacio de direcciones virtuales de memoria.

3.2.1 Estructura de cliente visible de una tarea

Una tarea es accesible mediante su puerto de kernel. Dicho puerto es accesible por la propia tarea, los hilos de la tarea, y cualquier otra tarea que posea derechos de escritura en ese puerto.

3.2.2 Creación y destrucción de tareas

Una nueva tarea se crea mediante el mensaje:

```
task->task_create
```

La pregunta es: ¿a quien enviamos este mensaje?. En Mach, el encargado de crear a una tarea es siempre otra tarea, a la que se denomina *tarea prototipo*. Se da una relación de herencia entre una tarea y su prototipo. La nueva tarea se ejecuta en la misma máquina que su tarea prototipo (y no en la de la tarea que envía el mensaje `task->task_create`).

En concreto, el estado de una tarea recién creada es:

- Espacio de direcciones de memoria virtuales vacío, o bien heredado de su tarea prototipo.
- Espacio de nombres de puertos vacío.
- Ningún thread.

`task->task_create` retorna un derecho de escritura en el puerto de kernel de la tarea recién creada.

Al igual que en el caso de los hilos, el puerto de kernel de la tarea sirve para que objetos externos a la tarea se refieran a ella misma. De '`task_special_ports.h`':

```
#define TASK_KERNEL_PORT 1 /* Represents task to the outside  world.*/
```

Una tarea se destruye mediante

```
task->task_terminate
```

NOTA: el mensaje task_terminate se envía a la tarea que se quiere terminar, y no a su tarea prototipo.

3.2.3 Puertos especiales de una tarea

Aparte de su espacio de nombres de puerto, las tareas poseen un pequeño conjunto de puertos especiales. Se dividen en dos grupos:

- Puertos especiales.
- Puertos registrados.

3.2.3.1 Puertos especiales

Las tareas disponen de tres puertos especiales:

- Un puerto para efectuar operaciones sobre sí misma. Al igual que en el caso de los hilos, este puerto suele ser el puerto de kernel de la tarea, aunque no es obligatorio.
- El puerto de excepciones de la tarea. Este puerto se utiliza cuando un hilo genera una excepción y no tiene definido ningún puerto de excepciones.
- El puerto de bootstrap Puede usarse para cualquier cosa, pero suele ser el primer puerto que posee una tarea a otro objeto aparte de ella misma. Por tanto, se utiliza para obtener puertos a objetos que ofrecen servicios. Por eso se llama "puerto de bootstrap".

Los tres puertos especiales (self, excepciones y bootstrap) pueden obtenerse mediante:

```
task->task_get_special_port
```

y fijarse mediante:

```
task->task_set_special_port
```

Los valores iniciales de estos puertos se heredan de la tarea prototipo correspondiente (con la excepción del puerto self, naturalmente :)

3.2.3.2 Puertos registrados

Las tareas también contienen un pequeño array de "puertos registrados". Estos puertos también se heredan de la tarea prototipo. Los puertos registrados pueden utilizarse para cualquier cosa, y en definitiva ofrecen puertos automáticamente heredados. Cosa muy útil.

Para *registrar* puertos se utiliza:

```
task->mach_ports_register
```

y para obtenerlos:

```
task->mach_ports_lookup
```

Es de notar que no se debe abusar de los puertos registrados. La forma general de almacenar derechos sobre puertos es utilizando el espacio de nombres de puertos de la tarea. Además, para ahuyentar toda tentación:

```
#define TASK_PORT_REGISTER_MAX 3
```

es decir, que solo disponemos de tres puertos registrados para cada tarea. Esto se debe a que la introducción de puertos registrados en mach se debió a ciertos motivos de

"conveniencia". De hecho, tradicionalmente estos tres slots contenían derechos de escritura a un puerto del servidor de "nombres de red", al servidor de "entorno", y al servidor de "servicios". Dado que uno puede organizar su sistema-encima-de-mach como le de la gana, al final se aplica lo mismo al uso de los tres puertos registrados. Como ejemplo, en Hurd (que es un sistema multiservidor que corre encima de mach) no se dispone de ninguno de esos servidores, y los puertos registrados se usan para otras cosas.

Los puertos registrados, pues, son algo bastante feo en mach: están hechos a medida para tener sistemas que tienen un servidor de nombres de red, otro de entorno, y otro que nombra servicios.

El Buen Modo(TM) de hacer las cosas es utilizar el puerto especial de bootstrap para obtener derechos de escritura (o lectura) de puertos de entidades externas, y almacenar esos derechos en el espacio de nombres de puertos de las tareas.

3.2.4 Operaciones sobre todos los hilos de una tarea

Dado que las tareas contienen hilos, es posible invocar sobre ellas operaciones que afectan a todos sus hilos, en masa. Dichas operaciones son similares a las que hemos visto en los propios hilos.

Veamos estas operaciones en masa son:

`task->task_suspend/task->task_resume`

: Suspende o despierta a todos los hilos de la tarea.

  Es de notar que esta operación no afecta a los contadores de suspensión de los hilos, sino al de la tarea.

  Un hilo, pues, puede ejecutarse si tanto su contador de suspensión como el de la tarea que la contiene son cero.

Operaciones de emulación de llamadas al sistema

: Como ya vimos en los hilos:

  - `task->task_set_emulation`
  - `task->task_set_emulation_vector`
  - `task->task_get_emulation_vector`

  `task->task_set_emulation` asigna a una rutina como tratamiento de una llamada al sistema dada.

  Es importante comprender que el ámbito de esta "emulación" es la tarea misma. Las indirecciones `llamada-al-sistema->rutina` fijadas con `task->task_set_emulation` solo tienen validez para los hilos contenidos en la tarea.

  Un ejemplo: si tenemos un sistema monoservidor (que consta de una sola tarea mach) que emula un sistema DOS, entonces deberemos fijar una rutina que atienda la *interrupción 21H* (indirección `21->tratar_interrupción_DOS`). En este sistema, pues, cuando un hilo de la tarea hace un trap 21, lo atiende esa rutina.

  Sin embargo, si nuestro sistema DOS sobre mach es multiservidor (consta de varias tareas mach), entonces deberemos "propagar" esa emulación a todas las tareas, de modo que cualquier hilo de cualquier tarea vea atendido un trap 21 por la rutina correspondiente (que a su vez debería estar propagada a todas

las tareas. Una opción mas elaborada sería tener una tarea que aceptara un mensaje para ejecutar la interrupción 21 de DOS. La indirección en el resto de las tareas sería entonces: `21->rutina_que_envia_mensaje_a_tarea_de_int_21_para_que_la_ejecute`). De todos modos, como suele ocurrir con los ejemplos, este ejemplo es un poco tonto, ya que emular un sistema tan simple como DOS con muchas tareas sería como matar moscas a cañonazos (aunque siempre podrías tener muchos de esos DOSES monoservidores rondando por ahí, junto con Hurd. Una especie de DOSEMU parásito :)

3.2.5 Parámetros de planificación de todos los hilos de la tarea

Al igual que las operaciones de parámetros de planificación sobre hilos individuales, tenemos las siguientes que modifican los parámetros de todos los hilos de una tarea.

- `task->task_assign`
- `task->task_assign_default`
- `task->task_get_assignment`
- `task->task_prority`

3.3 Algunas cuestiones importantes acerca de Tareas e Hilos

- Los puertos de kernel, tanto de las tareas como de los hilos, sirven para que otras tareas o hilos se refieran a ellos (les manden mensajes) . En cierto modo, puede verse la colección de puertos de kernel de tareas como una base de datos mantenida por el kernel de todas las tareas existentes en el sistema. Lo mismo es aplicable a los hilos.

  Una norma fundamental: si una tarea/hilo existe en el sistema, entonces existe un puerto de kernel para la tarea/hilo. Otro cantar es poseer derechos de escribir en esos puertos. Cualquiera que posea esos derechos puede enviar un mensaje a la tarea/hilo.

  En definitiva, los puertos de kernel de las tareas y los hilos identifican a las tareas y los hilos para el resto del sistema.

- Los puertos especiales 'self', tanto de las tareas como de los hilos, *suelen ser* los puertos de kernel. Esto no es obligatorio. Ya hemos visto que la finalidad de los puertos de kernel es que otras entidades puedan enviar mensajes. Parece lógico (y casi siempre se hace así) que una tarea/hilo también utilicen su puerto de kernel para referirse a sí misma. Pero mach da mas juego: una tarea prototipo podría "filtrar" las referencias a sí misma de la tarea hija. Posibilidades, infinitas...

- El sistema de creación de tareas (mediante una tarea "prototipo") puede ser confuso. En particular, NO es igual que las estructuras árbol de los procesos UNIX, aunque a primera vista lo parezca.

  En UNIX, los procesos se estructuran en forma de árbol, con relaciones padre->hijo* (un padre puede tener cero o mas hijos). Los hijos heredan determinadas cosas del padre: estado de procesador y secciones de datos y texto (datos en copia y texto compartido en solo lectura, habitualmente). Además, y esto es importante, a los hijos los crea el padre.

  En el caso de mach, la cosa es ligeramente distinta. Aun cuando las tareas reflejan una estructura en árbol, mediante la relación `prototipo->hija*` (una tarea puede ser

prototipo de cero o más tareas), y heredan determinadas cosas del padre (puertos especiales, registrados y regiones de memoria virtual), NO tiene que ser la tarea prototipo la que crea a la tarea hija. Cualquier tarea puede enviar un mensaje a otra para crear una nueva tarea a imagen y semejanza de la tarea llamada.

Una diferencia fundamental: en UNIX, todo proceso debe tener un padre, y además ese padre debe estar vivo. En Mach, sin embargo, puedes destruir alegremente una tarea prototipo, que los hijos ni se enteran.

En resumen: una tarea puede tener un hijo enviándose un mensaje `task->task_create` a sí misma. Pero también puede pedir hijos a cualquier otra tarea, poniéndola como destinatario del mensaje.

Está claro que las tareas mach son mucho mas promíscuas que los procesos UNIX (aunque esto no es difícil, ya que los procesos UNIX son monoparentales y hermafroditas).

- En cuanto a creación de tareas e hilos, acude esta pregunta: Si una tarea se crea a partir de otra... ¿de donde co*o sale la primera tarea?. Si no hay ninguna tarea, ¿a quién se le envía `task->task_create`?

  Al igual que en UNIX existe un proceso primigenio y abuelo por vocación llamado init, que no tiene padre, en los sistemas mach existe una tarea también abuela por vocación, que no tiene tarea prototipa: se la llama tarea bootstrap. Cuando un mach arranca, crea esa tarea. Habitualmente, la tarea bootstrap se encarga de poner en marcha el resto de las tareas que conforman el sistema que tengamos sobre mach (a menudo guiada por un fichero de configuración en disco). A diferencia de en UNIX, donde el abuelo es bastante siniestro (espera a que mueran todos los hijos) la tarea bootstrap puede morir sin remordimientos una vez creadas las tareas del sistema.
- El puerto bootstrap de una tarea puede ser cualquiera que le venga en gana a la tarea prototipo.

4 Escribiendo servidores con Mach y MIG

4.1 IPC vs. RPC

Las tareas Mach se comunican entre sí mediante mensajes. El kernel pone en manos de las tareas un conjunto de herramientas y abstracciones que hacen posible dicha comunicación:

- Las estructuras que forman los mensajes (mach_msg_header_t, etc).
- Las estructuras que identifican puertos (mach_port_t, etc).
- Una primitiva de envío de mensajes. En concreto, la rutina `mach_msg`.

Todo ello conforma lo que se llama el sistema de IPC (InterProcess Communication) de Mach. Utilizando el IPC de Mach, las tareas pueden comunicarse entre sí enviando mensajes,que previamente han montado, a puertos de otras tareas, via `mach_msg`.

Desgraciadamente, cuanto mas flexible es el sistema de mensajería mas complicados se vuelven los mecanismos de IPC. Si tenemos en cuenta que el sistema de mensajería que exporta Mach es notoriamente flexible y completo, podremos deducir que la escritura de tareas Mach que se comunican entre sí no es simple.

De hecho, no es en absoluto sencillo. La rica semántica de `mach_msg` se traduce en siete parámetros, la mayoría de ellos consistentes a su vez en estructuras complejas y enrevesadas. Los propios mensajes son un buen ejemplo: hay que montarlos y ajustarlos al tipo de envío, especificar de forma explícita campos como el tamaño de los datos contenidos en el mensaje, su número, sus tipos, el modo en que se gestiona la memoria que ocupan, etc.

Ante este panorama, se ideó una forma de ocultar la mayor parte de la complejidad que representa el sistema de IPC, basándose en que, si bien la flexibilidad en el sistema de mensajería es muy útil para manejar casos especiales y concretos, la mayoría de las veces basta con un subconjunto muy pequeño de todas las posibles variaciones. Es decir, que en la práctica se envían mensajes de dos, tres, o cuatro formas distintas, aunque el sistema de IPC nos permita hacerlo de muchas formas mas.

La estrategia consiste en ocultar el sistema de IPC debajo de una interfaz basada en RPC (Remote Procedure Call). Los sistemas RPC son muy utilizados en sistemas distribuídos como rpcsun o CORBA. En definitiva, se consigue que una tarea pueda invocar un procedimiento local como si fuera "remoto" (es decir, situado en otra tarea). El sistema de invocación de la rutina es idéntico al de cualquier otra rutina, y su semántica es exactamente igual. Veamos un ejemplo. Supongamos que una tarea (tarea1) quiere enviar dos números enteros a otra (tarea 2) para que los sume, y le retorne el resultado.

Utilizando IPC directamente, la tarea1 tendría que hacer lo siguiente:

```
-- tarea1 (IPC)

  /* Obtener de algun modo un derecho de envío a un puerto de tarea 2 */

  /* Obtener un derecho de envío a un puerto de tarea 1 para que sirva
   * de puerto de respuesta */
```

```
    /* Preparar los dos enteros para su viaje en el mensaje
     *
     * (en estructuras mas complejas puede ser trabajoso)
     *
     * MARSHALING
     */

    /* Montar el mensaje a enviar a tarea2 */

      /* Especificar en el mensaje que el destinatario es tarea2 */
      /* Indicar que el mensaje contiene dos enteros inline */
      /* Indicar las propiedades de estado de memoria de los enteros */
      /* Introducir en el mensaje el puerto de kernel de tarea1 */
      ... etc ...


    /* Enviar el mensaje mediante 'mach_msg' */

    /* Procesar los distintos posibles códigos de retorno de 'mach_msg'
     * (error, no caben mas mensajes, mensaje rechazado, y un largo etc)
     */


    /* Recibir el mensaje con la respuesta mediante 'mach_msg' */

    /* Procesar los distintos posibles códigos de retorno */

    /* Si todo ha ido bien, procesar el dato de respuesta (la suma).
     *
     * UNMARSHALING
     */

  ---
```

El lado de la tarea que suma ambos números sería similar

```
  -- tarea2 (IPC)

     /* Obtener puertos, recibir mensaje, unmarshalling, etc etc */

     ...

     suma = num1 + num2;
```

```
    ...

    /* Marshalling, enviar respuesta, tratar casos de error, etc etc */

---
```

Por contra, si utilizamos un sistema de RPC, el trabajo de la tarea1 consistiría en:

```
-- tarea1 (RPC)

    ...

    estado = sumadosnumeros(num1, num2, resultado_suma);

    ...

---
```

y lo que es mejor, el lado de la tarea que suma los números consistiría en:

```
-- tarea2 (RPC)

kern_return_t sumadosnumeros (int num1, int num2, int *resultado_suma)
{

resultado_suma = num1 + num2;

        return KERN_SUCCESS;
}

---
```

concentrándonos de ese modo en el trabajo realmente importante de la tarea, que en este caso es sumar los dos números.

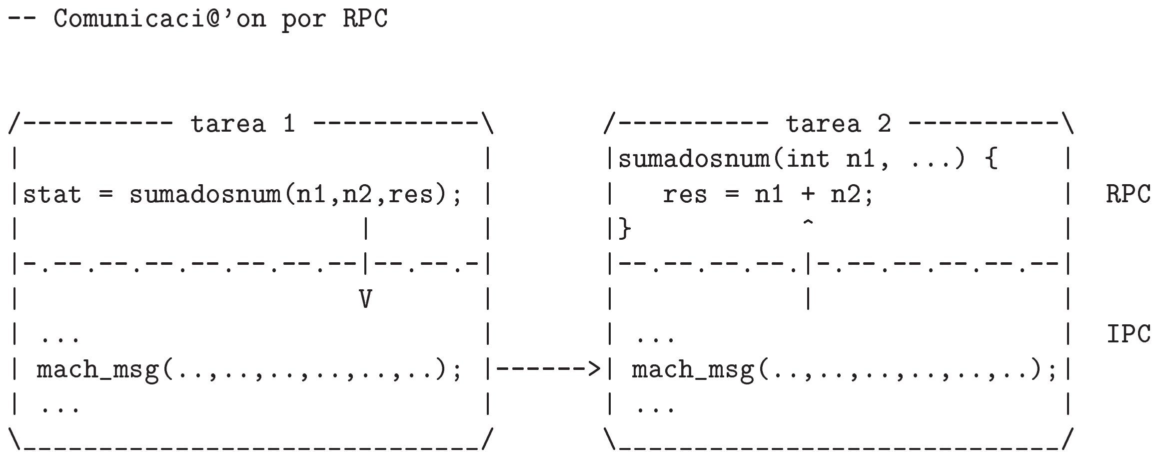
Sorprendente, no? Ninguna de las implementaciones de las tareas sabe nada de mensajes, puertos, errores de envío, marshalling y unmarshalling, etc etc.

Evidentemente (los milagros no existen) la implementación de 'sumadosnumeros' en la tarea1 deberá pelearse con el IPC, y al final enviar un mensaje mediante `mach_msg` y esperar la respuesta. De forma similar, algo en la implementación de tarea2 deberá utilizar IPC para recibir el mensaje, extraer los números a sumar, invocar finalmente a la rutina 'sumadosnumeros' del servidor, recojer el resultado y, mediante mas IPC, enviárselo de vuelta a tarea1.

Afortunadamente, esas partes tan desagradables pueden generarse automáticamente mediante un "generador de interfaces". Mach utiliza MIG (Mach Interface Generator) para, partiendo de un fichero en el que se especifica la interfaz que presenta una tarea (los RPC que "exporta") genera el código que se encarga del IPC, dejando al desarrollador libre para concentrarse en escribir lo importante: lo demás. En este documento aprenderemos a utilizar MIG para escribir servidores Mach.

Podemos ver esto gráficamente:

`-- Comunicaci@'on por RPC`




```
     Las partes inferiores son generadas por el generador de
     interfaces.

---
```

Una última nota: la abstracción RPC es extremadamente útil para la mayoría de las situaciones. Sin embargo, la flexibilidad de los sistemas RPC depende directamente de la capacidad expresiva de los generadores de interfaces, que puede ser mayor o menor. Un sistema IPC puede ofrecer mensajería asíncrona u otra potente característica y sin embargo el código generado por un generador de interfaces no hacer uso de esas funcionalidades. Cuando nuestro generador de interfaces no sea capaz de expresar algo que queremos programar tendremos que bajar a los infiernos (¡aunque excitantes!) del IPC, pero en la práctica esto no es muy frecuente, y generalmente seremos muy felices dejando que MIG combata a los demonios del IPC de Mach por nosotros.

4.2 Modelo cliente-servidor

Los sistemas construidos sobre Mach siempre tienden a seguir un modelo "cliente-servidor".

A una tarea que exporta servicios se la llama "servidor", y a una tarea que utiliza los servicios de otra se la llama "cliente". A menudo una tarea es tanto cliente como servidor: exporta unos servicios y utiliza otros de distintas tareas.

Los servicios que exporta un servidor determinan lo que se conoce como la "interfaz" del servidor. La interfaz determina un "lenguaje común" o protocolo. Los clientes que pidan algún servicio al servidor deberán hacerlo mediante ese protocolo y no otro. Si utilizamos RPC, entonces podemos implementar cada servicio como un procedimiento remoto que el servidor pone a disposición de los potenciales clientes. De este modo, la interfaz de un servidor se compone de los procedimientos remotos que exporta a las tareas clientes.

Como ejemplo de un sistema que sigue el modelo cliente-servidor, podemos imaginar un sistema construido sobre Mach en el que varias tareas colaboran para formar un sistema de cálculo. Nuestro sistema consiste en dos tareas:

- Una tarea encargada del cálculo: `tarea-cálculo`

  Esta tarea sabe sumar, restar, resolver integrales triples, etc.

  Exporta los RPC:

  - `sumar(num1,num2);`
  - `restar(num1,num2);`
  - etc...

- Una tarea encargada de pintar en la pantalla: `tarea-pintora`

  Esta tarea sabe pintar, y evidentemente tiene acceso a algún dispositivo físico como una pantalla o un LED.

  Exporta los RPC:

  - `pintar_caracter(c);`
  - `borrar(n);`
  - etc ...

En este ejemplo, `tarea-pintora` es servidor, porque `tarea-cálculo` utiliza sus servicios de pantalla para mostrar resutados de cálculo. Por su parte, 'tarea-cálculo' también es servidor porque otras tareas utilizan sus servicios de cálculo (podría usarlos la propia `tarea-pantalla` para calcular el número de caracteres que hay en una línea de la pantalla, por ejemplo).

De este modo, podemos modelizar los sistemas que componen nuestras tareas Mach en modelos cliente-servidor. Aparte de facilitar el diseño del sistema, nos permite obtener una visión abstracta de la modularidad y de las capacidades que conforman el sistema.

Como veremos, los ficheros de MIG definen interfaces, que a su vez definen el servicio que una tarea exporta a las demás.

En definitiva, utilizando el modelo cliente-servidor y las capacidades de MIG, podemos construir tareas Mach que copeeren entre sí de una forma sencilla, clara, racional y estructurada, huyendo así del caos que supone la utilización ad-hoc del IPC.

Un ejemplo excelente de sistema compuesto de tareas servidores y tareas clientes es HURD, el reemplazo para el núcleo de UNIX del Proyecto GNU.

4.3 El generador de interfaces MIG

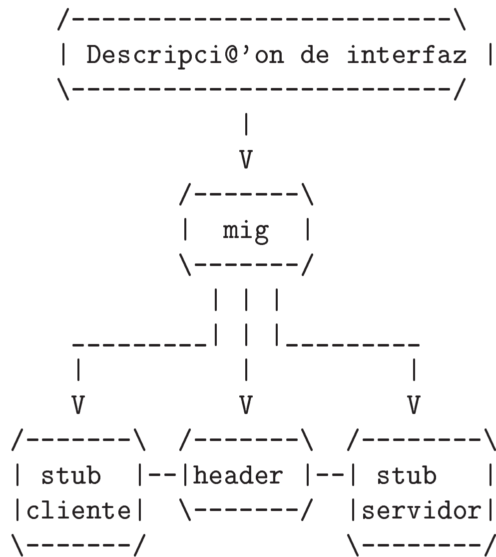
MIG (Mach Interface Generator) es el sistema de generación de interfaces de Mach. Como hemos visto, permite abstraernos del IPC y construir nuestros sistemas multi-tareas de una forma estructurada.

Aquí aprenderemos a utilizar MIG de forma básica, lo que nos permitirá escribir sistemas compuestos de varias tareas, de interfaces bien definidas.

4.3.1 Operativa básica

La operativa de MIG está reflejada en el siguiente diagrama:

`-- Operativa de MIG`



`---`

Donde *mig* (un programa), recibe como entrada un fichero de definición de interfaz, escrito en un lenguaje diseñado específicamente para definir interfaces de tareas Mach. El lenguaje MIG es un ejemplo concreto del concepto mas general de *lenguajes IDL* (Interfaze Definition Languages). Los ficheros de definición de interfaces de MIG suelen tener extensión '`.defs`'. Por ejemplo, '`clock.defs`'.

A partir de un fichero de interfaz (como los denominaremos de aquí en adelante) MIG genera tres ficheros de salida:

- El *stub* del cliente (por ej. '`clock_client.c`'), que contiene el código necesario para poder ofrecer a la tarea cliente una RPC por cada uno de los procedimientos remotos de que se compone la interfaz. Por ejemplo, 'reloj.defs' podría definir los servicios `ajustar_hora` y `leer_hora`. En ese caso, `clock_client.c` contendría la definición de los procedimientos 'ajustar_hora' y `leer_hora`. Serán esos procedimientos los que se encarguen de manejar la IPC de Mach para enviar mensajes a un servidor que implemente la interfaz *clock*, y obtener sus resultados.

- El *stub* del servidor (por ej. '`clock_server.c`'), que contiene el código necesario para poder recibir mensajes de una tarea cliente mediante la interfaz *clock*. El código de stub del servidor se encargará de manejar la IPC de mach para recibir mensajes desde tareas clientes que utilicen la interfaz *clock*, e invocar a los procedimientos `leer_hora` y `ajustar_hora` implementados en el servidor por el desarrollador.
- Un fichero de cabecera C con definiciones comunes a los stubs de cliente y de servidor.

Podemos por tanto ver la operativa en forma de ficheros:

```
-- Operativa de MIG (ficheros)


                     --> clock_client.c
                    /
 clock.defs --> mig  ---> clock.h
                    \
                     --> clock_server.c

---
```

Y ya tenemos generado el código que implementa la interfaz *clock*. Ahora, podemos utilizar estos stubs en cualquier tarea que "hable en clock", tanto como cliente como en calidad de servidor. Esto es importante: la relación entre tareas e interfaces no es 1-1. De hecho, es común que existan tareas servidores que hablen en mas de un idioma, es decir, que implementen varias interfaces.

Si queremos escribir un servidor `servidor_hora` que implementa el protocolo *clock* definido anteriormente, y un pequeño cliente de prueba que pregunte la hora y luego la fije, podemos utilizar la siguiente estructura de ficheros fuente:

```
-- Ficheros del proyecto 'hora'

   cliente_de_prueba.c   -> C@'odigo del cliente de prueba

   servidor_hora.c       -> C@'odigo del servidor hora.


   clock_client.c   \
   clock_server.c   |-> Protocolo 'clock' generado por MIG.
   clock.h          /


----
```

'`servidor_hora.c`' implementará una función `main` con el bucle de servicio (que veremos mas adelante) y por supuesto la implementación de las rutinas `fijar_hora` y `obtener_hora`, que implementan los servicios del protocolo *clock*.

'`cliente_de_prueba.c`', por su parte, contendrá un pequeño programa que invoque a los procedimientos `fijar_hora` y `obtener_hora`.

A la hora de construir los programas, se enlazará el código de cliente ('`cliente_de_prueba.c`') con el stub de cliente generado por MIG ('`clock_client.c`'),

y el código de servidor (‘servidor_hora.c’) con el stub de servidor generado por MIG (‘clock_server.c’). Es decir:

```
-- Construcci@'on de las tareas

    servidor_hora.c  \
                     |-> gcc -> servidor_hora*
    clock_server.c   /    ^
                          |
                          |
                        clock.h
                              |
                              |
                              |
    cliente_de_prueba.c  \    V
                         |-> gcc -> cliente_hora*
    clock_client.c       /


---
```

Primero aprenderemos a escribir ficheros de definición de interfaz, paso a paso. Como caso práctico, escribiremos el fichero de definición del protocolo 'clock'. La escritura de 'cliente_de_prueba.c' y 'servidor_hora.c' la dejaremos para el siguiente capítulo.

4.3.2 Estructura de un fichero de definición de interfaz

Un fichero de definición de interfaz MIG se compone de las siguientes partes:

- Identificación de subsistema.
- Definición de tipos.
- Definición de servicios que componen la interfaz.

Veamos cada una por partes, despacio y con buena letra.

4.3.2.1 Subsistemas

En nomenclatura MIG las diferentes interfaces se denominan *subsistemas*. Esto sigue el concepto de que un conjunto de tareas que cooperan entre sí forman un "sistema". Esto puede resultar confuso ya que, como hemos dicho antes, una interfaz puede ser utilizada por varias tareas. Yo encuentro bastante esclarificador el denominar a las interfaces "protocolos", ya que es un término utilizado en otros ámbitos como las redes, y que refleja el mismo concepto: un *idioma común*. A lo largo del documento utilizaré las palabras *subsistema*, *interfaz* y *protocolo* indistintamente.

Cada subsistema MIG posee un nombre y un número entero que lo identifica en todo el sistema. El identificador de subsistema debe ser único entre todos los subsistemas utilizados en un sistema de tareas Mach. Veremos mas adelante que esto sirve para que las rutinas demultiplexoras de los stubs de servidor identifiquen qué servicio se está requiriendo en un mensaje. Por ahora baste saber que este número debe ser único entre todas las interfaces. Afortunadamente, puede ser bastante grande. Es costumbre que tengan siete cifras.

Lo primero que debemos poner en un fichero MIG es la definición del subsistema que estamos definiendo. El formato es el siguiente:

`subsystem` *`nombre id`* `;`

Donde *nombre* es el nombre del subsistema, e *id* es el identificador del subsistema (un entero). En el caso de nuestra interfaz *clock*, pondremos:

```
subsystem clock 5000000;
```

4.3.2.2 Tipos

En los ficheros MIG necesitamos poder expresar tipos de datos. Su utilidad básica es la de expresar los tipos de los parámetros de los servicios, como veremos mas adelante. Al igual que otros muchos lenguajes, MIG proporciona una serie de tipos primitivos (que llamamos *tipos IPC*) y facilidades para crear tipos nuevos. Lo nuevos tipos pueden ser tanto anónimos como nombrados. Además, podemos crear tipos básicos y estructurados. Veamos cómo.

Una definición de tipo tiene la forma:

`type` *`nombre-nuevo-tipo`* `=` *`descripción-tipo`* `[`*`info-de-traducción`*`]`

(lo encerrado entre corchetes [] es opcional)

Donde,

nombre-nuevo-tipo
: Es el nombre del tipo que estamos definiendo.

descripción-tipo
: Es la combinación de otros tipos, que forman al nuevo tipo.

info-de-traducción
: Indica los tipos C al generar los stubs. No lo utilizaremos.

descripción-tipo
: Puede tener dos formas:
  - Un tipo de usuario previamente declarado.
  - Un tipo IPC.

Los tipos IPC básicos tienen dos formas:

`(`*`nombre-tipo-ipc`*`,` *`tamaño`* `[,`*`flags IPC`*`])`

o

`nombre-tipo-ipc`

Donde, *nombre-tipo-ipc* es el nombre del tipo IPC, *tamaño* es la longitud en bits del tipo IPC, y *flags IPC* son una serie de flags separados por comas, que definen algunos aspectos IPC del tipo. Los flags soportados son:

dealloc - dealloc [] - notdealloc - islong - isnotlong

Sirven para definir cuestiones de marshalling y unmarshalling, y de almacenamiento. A nivel básico no se utilizan.

Entre otros, disponemos de los siguientes tipos IPC básicos:

```
-- Tipos b@'asicos IPC

- MACH_MSG_TYPE_CHAR
```

- MACH_MSG_TYPE_INTEGER_16

- MACH_MSG_TYPE_INTEGER_32

- MACH_MSG_TYPE_BOOLEAN

- MACH_MSG_TYPE_STRING

MACH_MSG_TYPE_STRING solo puede expresarse en la forma '(nombre-tipo, tama@~no)'.

Algunos ejemplos de declaraciones de tipos básicos:

```
type int = MACH_MSG_TYPE_INTEGER_32;

type mi_cadena = (MACH_MSG_TYPE_STRING, 8*80);

type kern_return_t = int;
```

Los tipos IPC estructurados se forman mediante *arrays* y *estructuras*. Las formas de definirlos son:

struct [tamaño] of *descripción-tipo*

array [tamaño] of *descripción-tipo*

de forma similar a los lenguajes tipo Algol como Pascal.

Por último, están los tipos *puntero* (o *fuera de línea*). Se construyen con el operador tilde:

^*descripción-tipo*

Sirven para indicar de forma explícita que los datos de este tipo irán en el mensaje IPC subyacente como *fuera de línea*. Esto significa que el mensaje contendrá un puntero a los datos, no los datos en sí mismos.

Veamos algunos ejemplos de definición de tipos IPC estructurados:

```
-- Ejemplo de tipos estructurados
```

```
    type procids  =      array [10] of int;

    type procidinfo =    struct [5*10] of MACH_MSG_TYPE_INTEGER_32;

    type array_por_valor = struct [1] of array [20] of
                                MACH_MSG_TYPE_CHAR;

    type pagina_ptr =     ^array[4096] of MACH_MSG_TYPE_INTEGER_32;

  ---
```

Vemos que los tipos IPC son algo aparatosos. De hecho, no se suelen utilizar directamente, y solo aparecen en las definiciones de otros tipos. Afortunadamente, en Mach disponemos del fichero '`std_types.defs`', que podemos *importar* a nuestros ficheros de interfaz. Importar un fichero .defs desde otro es algo similar al #include del preprocesador de C, y de hecho tiene la misma sintáxis y semántica.

'`std_types.defs`' define multitud de tipos que tienen contrapartida C directa. Es realmente extraño utilizar otros tipos básicos que no estén en '`std_types.defs`' y '`mach_types.defs`' (otro fichero que deberemos importar casi siempre).

Tipos útiles de '`std_types.defs`' y '`mach_types.defs`':

```
/* extracto de 'std_types.defs' */


type char = MACH_MSG_TYPE_CHAR;
type short = MACH_MSG_TYPE_INTEGER_16;
type int = MACH_MSG_TYPE_INTEGER_32;
type int32 = MACH_MSG_TYPE_INTEGER_32;
type boolean_t = MACH_MSG_TYPE_BOOLEAN;
type unsigned = MACH_MSG_TYPE_INTEGER_32;
type unsigned32 = MACH_MSG_TYPE_INTEGER_32;
type int64 = MACH_MSG_TYPE_INTEGER_64;
type unsigned64 = MACH_MSG_TYPE_INTEGER_64;
#include <mach/machine/machine_types.defs>

type kern_return_t = int;

type pointer_t = ^array[] of MACH_MSG_TYPE_BYTE
ctype: vm_offset_t;
```

Por tanto, en '`clock.defs`' pondremos justo después de la definición de subsistema:

```
   #include <mach/std_types.defs>
```

```
#include <mach/mach_types.defs>
```

4.3.2.3 Servicios

Por fin algo de *chicha* en nuestro fichero MIG. Ahora definiremos los servicios que conforman nuestro protocolo.

Cada definición de servicio tiene la forma:

```
tipo-servicio nombre-servicio
(

   modificador-parámetro nombre-parámetro : tipo-parámetro;
   ...
   ...
);
```

Donde,

tipo-servicio
: Especifica el tipo de servicio.

  Puede ser `routine`, para especificar un servicio síncrono (que espera una respuesta del servidor), o `simpleroutine` para especificar un servicio asíncrono (que no espera respuesta del servidor).

nombre-servicio
: Es el nombre del servicio. Será el nombre del RPC que llamaremos desde el cliente, y el nombre de la rutina a implementar en el servidor.

modificador-parámetro
: Puede ser uno o varios de:

  in - out - inout - requestport - replyport - sreplyport - ureplyport - waittime - msgseqno - msgoption

nombre-parámetro
: Es el nombre del parámetro.

tipo-parámetro
: Es el tipo del parámetro. Debe tener una contrapartida C.

Como ejemplo veamos las definiciones de servicio de nuestro protocolo *clock*:

```
routine fijar_hora
(

     reloj       :   reloj_t;
     nueva_hora  :   hora_t
```

```
);


routine obtener_hora
(

     reloj       :  reloj_t;
out  hora        :  hora_t

);
```

FIXME: Completar.

4.4 Mas sobre IDs, y skip;

TODO: Escribir.

4.5 Un servidor Mach simple: servidor_hora

El programa principal de nuestro ‘servidor_hora.c’ puede escribirse así:

```
int main()
{


  /*
   * Obtener un puerto de servicio en el que recibir peticiones de
   * servicio de los clientes.
   */

  mach_port_t puerto_de_servicio;

  (void)  mach_port_allocate (mach_task_self(),
                              MACH_PORT_RIGHT_RECEIVE,
                              &puerto_de_servicio);

  /*
   * Utilizar mach_msg_server para iterar sobre los mensajes
   * de petición de servicio, invocando en cada llegada la
   * rutina de demultiplexión de mensajes (normalmente generada
   * por MIG)
   */

  result = mach_msg_server (rutina_demultiplexora,/* Rutina
                                                   * demultiplexora */
                            MAX_MSG_SIZE,         /* Tamaño máximo de
```

```
                                                          los mensajes de
                                             respuesta */
                                 puerto_de_servicio);  /* Puerto en el que
        * recibir mensajes */

       return 0;

     }
```

La rutina `mach_msg_server` implementa el bucle de servicio del servidor. Recibe un mensaje reclamando un servicio, lo procesa invocando a la rutina de servicio apropiada, y envía un mensaje de respuesta mientras espera nuevos mensajes reclamando mas servicios.

Además, oculta muchos aspectos del uso de `mach_msg`, especialmente situaciones de error.

La rutina demultiplexora se invoca sobre cada mensaje de petición de servicio que se recibe en `service_port`. Retorna TRUE si el mensaje ha sido procesado por alguna rutina de servicio. En cualquier caso, monta el mensaje de respuesta. Esta rutina es normalmente *`interfaz`*`_server`, generada por MIG. Lo más importante de la rutina demultiplexora es que oculta los detalles del formato de los mensajes Mach (cabecera, etc). Las rutinas de servicio, pues, son invocadas por la rutina demultiplexora. De este modo, las rutinas de servicio (escritas por el usuario) pueden trabajar directamente con los datos contenidos en el mensaje de petición. La rutina demultiplexora generada por MIG ya ha extraído esos datos.

Los beneficios de utilizar `mach_msg_server` pueden resumirse:

– Eficiencia: utiliza send/receive combinado siempre que sea posible.
– Envía un mensaje de respuesta siempre que exista un puerto de respuesta (contenido en el mensaje de petición).
– Gestiona el espacio de memoria virtual ocupado por los mensajes.
– Efectúa cierto control de flujo (puertos de respuesta llenos, mensajes muy largos, etc).
– Efectúa control de errores (puertos de respuesta muertos, etc).

El resto del servidor consiste en la implementación de las rutinas de servicio propiamente dichas. Ahí es donde el desarrollador debe aplicar su arte.

Por ejemplo, dada la definición MIG de una rutina de servicio:

```
routine obtener_hora
(
      reloj          : mach_port_t;
      out hora       : hora_t
);
```

Y la llamada que hace un cliente:

```
estado = obtener_hora (reloj, &hora);
```

El desarrollador solo necesita proporcionar la siguiente rutina de servicio:

```
kern_return_t obtener_hora (mach_port_t reloj,
                            hora_t *hora)
{
   *hora = hora_actual;

   return KERN_SUCCESS;

}
```

Lo mismo para `fijar_hora`:

```
kern_return_t fijar_hora (mach_port_t reloj,
                  hora_t hora)
{


   hora_actual = hora;

   return KERN_SUCCESS;

}
```

Así de simple. Como vemos, el desarrollador puede abstraerse de las cuestiones relativas a puertos, mensajes, bucle de servicio, etc, y concentrarse en su trabajo real: el servicio proporcionado por las rutinas de servicio.

4.6 Un servidor que implementa varias interfaces MIG

La rutina demultiplexora *`servicio`*`_server` generada por MIG demultiplexa mensajes asociados con un fichero MIG (es decir, un subsistema). Por ejemplo, podemos definir una interfaz o subsistema *reloj* en un fichero MIG, que defina los siguientes servicios:

```
-- ‘reloj.defs’


subsystem reloj 3000000;

...

routine obtener_hora ( ... );       /* ID 300100 */
```

```
routine fijar_hora ( ... );         /* ID 300200 */

routine resetear_hora ( ... );      /* ID 300300 */

---
```

La rutina `reloj_server` generada por MIG en el stub del servidor será capaz de multiplexar una petición de servicio entre los tres servicios `obtener_hora`, `fijar_hora` y `resetear_hora`.

No obstante, a veces un servidor necesita manejar mensajes que pueden ser recibidos de varias interfaces, definidas en diferentes ficheros MIG. Por ejemplo, piénsese en una interfaz definida en el subsistema MIG *reloj_tr*, que defina servicios similares a los de *reloj*, pero con características de tiempo real:

```
-- ‘reloj_tr.defs’


subsystem reloj_tr 4000000;

...

routine obtener_hora_tr ( ... );       /* ID 400100 */

routine fijar_hora_tr ( ... );         /* ID 400200 */

routine resetear_hora_tr ( ... );      /* ID 400300 */

---
```

En el stub de servidor generado de esta interfaz estará la rutina demux `reloj_tr_server`, que será capaz de demultiplexar entre las tres rutinas `obtener_hora_tr`, `fijar_hora_tr` y `resetear_hora_tr`.

Ahora supongamos que estamos implementando un servidor llamado *reloj universal* que es capaz de gestionar relojes convencionales y relojes de tiempo real. Queremos que este servidor exporte tanto la interfaz *reloj* como la interfaz *reloj_tr* a futuros clientes. Como hemos visto, basta con que nuestro servidor implemente las rutinas de servicio de ambas interfaces:

`obtener_hora, obtener_hora_tr fijar_hora, fijar_hora_tr resetear_hora, resetear_hora_tr`

y enlazar en el servidor los stubs de servidor generados por MIG de ‘`reloj.defs`’ y ‘`reloj_tr.defs`’.

Pero se nos presenta un problema: ¿qué rutina demux ponemos en `mach_msg_server`? No podemos poner `reloj_server`, ya que sólo demultiplexaría las rutinas `*_hora`, y tam-

poco `reloj_tr_server`, ya que sólo demultiplexaría las rutinas `*_hora_tr`. Nuestro bucle de servicio debería poder atender peticiones de servicio para todos los servicios definidos por las dos interfaces.

En esos casos, hay que construir una rutina demultiplexora a medida que demultiplexe los servicios de todas las interfaces que implemente nuestro servidor. Llamaremos a esas rutinas demux *rutinas superdemultiplexoras o superdemux*.

Es importante que nuestras rutinas demultiplexoras sigan el mismo protocolo que las generadas por MIG, ya que es la única forma de que puedan trabajar con `mach_msg_server`. El protocolo es simple:

```
-- Protocolo 'mach_msg_server' <-> rutina demux

   * Debe recibir dos mensajes (tipo mach_msg_header_t *):

       - El mensaje de petición  (primer parámetro)
       - El mensaje de respuesta (segundo parámetro)

   * Debe retornar una de dos:

      TRUE  -> El mensaje ha sido procesado, y el mensaje de
               respuesta (segundo parámetro) está montado de forma adecuada,
               listo para enviarse.

      FALSE -> El mensaje no ha sido procesado debido a que reclamaba
               un servicio no ofrecido por el/los subsistemas que
               implementa el servidor.

---
```

En seguida acude a la mente: podemos simplemente invocar las diferentes rutinas demultiplexoras de cada servicio. Si el servicio pertenece a alguno de los sistemas, se llevará a cabo. Es tentador, así que elaboramos la primera versión de nuestra rutina superdemultiplexora:

```
boolean_t reloj_universal_demux (mach_msg_header_t *inmsg,
                                 mach_msg_header_t *outmsg)
{

   if (reloj_server(inmsg)    ||
       reloj_tr_server(inmsg))
   {
      return TRUE; /* El mensaje ha sido procesado
                    * por alguno de los <sistema>_server anteriores */
```

```
    }
    else
    {
       return FALSE; /* El mensaje no ha podido ser procesado */

    }

 }
```

La rutina anterior efectúa el trabajo correctamente: se invocan una tras otra las rutinas generadas por MIG. Si alguna de ellas ha procesado el mensaje (ha invocado a la rutina de servicio apropiada) retorna TRUE y la evaluación OR se detiene, evaluando a cierto. Si ninguna de ellas ha procesado el mensaje, implica que el servicio solicitado no es ofrecido en ninguno de los sistemas. En definitiva, estamos utilizando todas las rutinas demultiplexoras generadas por MIG.

No obstante, utilizar de este modo las rutinas demux de los subsistemas no es muy eficiente. Podemos aprovechar un conocimiento mas profundo de MIG para escribir una rutina superdemux mas fina y eficiente. Aprendamos algo mas de MIG, pues.

Las rutinas demux que genera MIG (`interfaz_server`) efectúan dos tareas:

1. Inicializa (monta) el mensaje de respuesta.
2. Si el mensaje de petición identifica un servicio implementado en el servidor, invoca a una rutina intermedia pasándola como parámetros el mensaje petición y el mensaje de respuesta (ya montado). Esta rutina intermedia (existe una por cada rutina de servicio) comprueba que los parámetros para la rutina de servicio sean correctos y los extrae del mensaje. Finalmente, invoca a la rutina de servicio del servidor, pasándole los parámetros adecuados. Podemos denominar a las rutinas intermedias *rutinas stub* de una rutina de servicio.

Existen dos rutinas que implementan *1.* y *2.* directamente.

La primera es `mig_reply_setup`, que recibe el mensaje de petición y retorna en un parámetro por referencia el mensaje de respuesta ya montado. `mig_reply_setup`, entonces, monta el mensaje de respuesta e implementa *1.* Esta rutina forma parte de la librería de MIG.

La segunda es `interfaz_server_routine`, generada por MIG en el stub del servidor. Es tan simple que pasteo aquí su esqueleto genérico:

```
 mig_external mig_routine_t subsistema_server_routine
                (mach_msg_header_t *InHeadP)
 {

       register int msgh_id;

       msgh_id = InHeadP->msgh_id - <num_subsistema>;
```

```
        if ((msgh_id > num_rutinas_del_subsistema ||
            (msgh_id < 0))
         {
            return 0;
         }

        return subsistema_subsystem.routine[msgh_id].stub_routine;

}
```

Vemos que *subsistema*`_server_routine` recibe como parámetro un mensaje de petición. Extrae en `msgh_id` el ID MIG que identifica a la rutina de servicio deseada en el stub del servidor. A continuación, utiliza el ID MIG para verificar que el servicio solicitado esté implementado en el subsistema. Si es así, retorna la rutina stub del servicio solicitado, que a su vez invocará a la rutina de servicio deseada.

Por tanto, *subsistema*`_server_routine` retorna o bien la rutina stub del servicio deseado, o bien cero (significando que el subsistema no atiende el servicio solicitado en el mensaje de petición).

Es de notar que las rutinas stub son de tipo `mig_routine_t`, al igual que todas las rutinas generadas por MIG en los stubs.

Armados ya con esos conocimientos sobre los stubs de servidor generados por MIG, ahora podemos reescribir nuestra rutina superdemultiplexora de una forma mas fina y eficiente:

```
boolean_t reloj_universal_demux2 (mach_msg_header_t *inmsg,
                                  mach_msg_header_t *outmsg)
{

   mig_routine_t rutina_stub;

   mig_reply_setup (&inmsg, &outmsg);   /* Inicializa el mensaje de
                                         * respuesta */

   if ((rutina_stub = reloj_server_routine(&inmsg) != 0) ||
       (rutina_stub = reloj_tr_server_routine(&inmsg) != 0))
   {

     /* Ahora routine apunta a la rutina stub del servicio solicitado */

     (*routine) (&inmsg, &outmsg); /* Invoca a la rutina stub
                                    * del servicio */

     return TRUE; /* Hemos procesado el mensaje */

   }
```

```
    else
    {
      /* Las rutinas server_routine de nuestros subsistemas han retornado
       * todas cero. Por tanto, el mensaje de petición no identifica
       * ningún servicio ofrecido por nuestros subsistemas */


      return FALSE; /* No hemos procesado el mensaje */

    }

}
```

4.7 Servicios Asíncronos

Las `routines` MIG definen interfaces síncronas a servicios. Es decir, el cliente, una vez hecha la petición de servicio, espera a que el servidor procese la petición y retorne un código indicando éxito o fallo de la operación. La principal ventaja de las interfaces síncronas es su claridad y eficiencia. En sistemas que requieren cierto paralelismo, no obstante, a menudo es útil disponer de servicios asíncronos, en los que el cliente formula su petición y, sin perder tiempo, continúa ejecutándose. Es posible que eventualmente el servidor envíe al cliente algún tipo de información resultado de la operación procesada, y es posible que no.

Pese a que los servicios asíncronos sean útiles en algunas ocasiones, presentan una serie de inconvenientes, algunos de ellos bastante serios:

- La programación asíncrona es difícil. Los programas asíncronos son también difíciles de mantener. Dado que tanto el cliente como el servidor utilizan RPC, la falta de sincronismo no es evidente (las invocaciones a rutinas convencionales siempre son síncronas). Eso puede llevar a programas difíciles de mantener, y por tanto tendentes a sufrir errores.
- Las implementaciones de servicios asíncronos a menudo son ineficientes. Por cierto que es así en Mach, principalmente pensado para utilizar mensajería síncrona.

En MIG se define un servicio asíncrono utilizando `simpleroutine` en lugar de `routine`. Añadamos un servicio síncrono llamado `fijar_factor_a` a nuestra interfaz *multiplicador*.

```
-- ‘multiplicador.defs’

...

routine fijar_factor_a (

     servidor     :  mach_port_t;
     factor       :  int
);
```

```
---
```

El usuario invoca este servicio como la contrapartida síncrona. Sencillamente:

```
resultado = fijar_factor_a (multiplicador, factor);
```

La rutina de servicio del servidor podría escribirse así:

```
kern_return_t fijar_factor_a (mach_port_t servidor,
                              int factor)
{

   multiplicador_factor = factor;

   return KERN_SUCCESS;

}
```

Sorprendentemente, las implementaciones son casi idénticas a las del servicio síncrono `fijar_factor`. La llamada de cliente todavía retorna un valor (indicando el éxito del envío del mensaje subyacente) y la rutina de servicio retorna otro (indicando al bucle de servicio que ha servido la petición de servicio).

Otra forma de asincronismo, mas compleja y que veremos en otra ocasión, ocurre cuando el servidor retorna (eventualmente) una respuesta al cliente. Un ejemplo sería una versión asíncrona de `obtener_factor`. Para ello se utiliza MIG de una forma mas avanzada, utilizando varios subsistemas para definir la interfaz del lado del cliente, del lado del servidor, y del propio mensaje de respuesta.

5 Ensamblador, PowerPC y Mach

5.1 Arquitectura PowerPC

5.2 Ensamblador PowerPC

5.3 Macros para escribir ensamblador en Mach: `asm.h`

6 Implementación de bloqueos en Mach

Entre otras cosas, las abstracciones del kernel ofrecen modos de sincronización de hilos, o control de concurrencia. El primer requisito para ello es disponer de cerrojos, para a continuación implementar con ellos las distintas primitivas de sincronización (como exclusiones mútuas, esperas activas (*spinlocks*), regiones críticas, etc).

Como era de esperar, la implementación de cerrojos es bastante sensible a la máquina en la que estemos ejecutando el sistema. Distintas arquitecturas hardware proporcionan distintas facilidades para implementar lo que constituye la característica fundamental necesaria para que un cerrojo funcione: la atomicidad en las operaciones que gestionan el cerrojo. Por otro lado, una vez disponemos de una implementación de los cerrojos, podemos construir el resto de primitivas de bloqueo de modo independiente de la máquina.

Por tanto, la implementación de bloqueos (o cerrojos) en Mach se divide en dos partes bien diferenciadas: la dependiente de la máquina (que implementa lo que se conoce como *hardware locks*, y la independiente de la máquina (que implementa el resto de tipos de bloqueos utilizando *hardware locks*).

6.1 La parte dependiente de la máquina

El TAD (Tipo abstracto de datos) que debe exportar la implementación de cerrojos hardware (y que es utilizado por el resto de la implementación de cerrojos) consiste en los tipos:

`hw_lock_data_t`
: Que representa los datos con los que la máquina implementa un cerrojo.

`hw_lock_t`
: Que es un puntero a una estructura `hw_lock_data`

y las operaciones:

- `hw_lock_init`
- `hw_lock_lock`
- `hw_lock_unlock`
- `hw_lock_try`
- `hw_lock_held`

que se explicarán luego.

Los ficheros que implementan los cerrojos hardware se encuentran, naturalmente, en el directorio de código dependiente de la arquitectura que estamos utilizando. En nuestro caso, en '`ppc/`'. Consisten en un fichero de cabecera que implementa los tipo `hw_lock_data_t` y `hw_lock_t` y un fichero escrito en ensamblador que implementa las operaciones del TAD.

6.1.1 Definición de tipos de bloqueos hardware: `ppc/lock.h`

El cometido principal de este fichero es definir los tipos requeridos por la parte dependiente de la máquina `hw_lock_data_t` y `hw_lock_t`.

Veamos la definición de `hw_lock_data_t`, que es una estructura:

```
struct slock {
```

```
int lock_data;
#if NCPUS > 1
int lock_padding[7];
#endif /* NCPUS > 1 */
};

typedef struct slock hw_lock_data_t;
```

Vemos que se utiliza un entero máquina (`lock_data`) para definir el estado de un cerrojo. En el caso de que dispongamos mas de un procesador, no obstante, es necesario un dato adicional: `lock_padding`, que consiste en un "relleno" de siete enteros máquina.

Por su parte, el tipo `hw_lock_t` se define simplemente como un puntero a una estructura `hw_lock_data_t`:

```
#define struct slock *hw_lock_t;
```

6.1.2 Implementación de las operaciones sobre bloqueos hardware: ppc/hw_lock.s

Recordemos las operaciones que debe implementar el sistema de cerrojos hardware:

`void hw_lock_init(hw_lock_t)`
: Inicializa el cerrojo hardware que se le pasa como argumento.

  La implementación es muy simple:

  ```
  li r0, 0 /* libera el cerrojo == 0 */
  stw r0, 0(r3) /* Inicializa el cerrojo con 0 */
  ```

  Es decir: almacena un cero en `lock.lock_data`, indicando así que el cerrojo está abierto.

`void hw_lock_lock(hw_lock_t)`
: Adquiere el cerrojo. Si el cerrojo ya está cerrado, espera de forma activa hasta que esté disponible.

`void hw_lock_unblock(hw_lock_t)`
: Libera el cerrojo de forma incondicional.

`unsigned int hw_lock_try(hw_lock_t)`
: Intenta obtener el spin-lock.

  Retorna éxito (1) o fallo (0), sin bloquearse.

`unsigned int hw_lock_held(hw_lock_t)`
: Retorna 1 si el cerrojo está cerrado. 0 en caso contrario.

Todas estas rutinas se implementan en 'ppc/hw_locks.c'. Por supuesto, están escritas en ensamblador y son 100% dependientes de la arquitectura hardware.

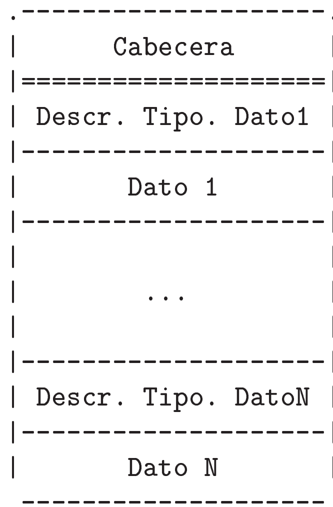
En el caso de la arquitectura PowerPC, 'ppc./hw_locks.c' también implementa las exclusiones mútuas. Como veremos, existe una implementación alternativa, independiente de la máquina, escrita en C. No obstante, se utiliza preferentemente la escrita en ensamblador, dependiente de la máquina, por motivos de eficiencia.xo

6.2 La parte independiente de la máquina

7 IPC de Mach

7.1 Estructura de Mensajes IPC

La estructura de un mensaje IPC puede verse en el siguiente diagrama:



Es decir, una cabecera seguida de pares `Descripción tipo dato - dato`, que describen los datos contenidos en el mensaje. El tipo del dato determina, entre otras cosas, la longitud del campo `dato` siguiente.

De todos esos campos, únicamente la cabecera es estrictamente necesaria para conformar un mensaje correcto, listo para enviarse.

7.2 Construyendo Mensajes IPC

La forma de construir mensajes IPC es definir una estructura C, que contendrá tantos campos como *elementos* contenga el mensaje. Por *elemento* nos referimos a uno de:

- Cabecera
- Descripción de tipo de dato
- Dato (que sigue a una descripción de tipo de dato)

Comencemos viendo la definición de una estructura que representa un mensaje compuesto únicamente por una cabecera:

```
struct
{
   mach_msg_header_t Cabecera;
};
```

Obsérvese que el mensaje definido por la estructura anterior contiene todo lo necesario para ser enviado: el puerto de destino, el puerto de réplica (si existe), etc, están contenidos en la cabecera (evidentemente, la cabecerá deberá “rellenarse” convenientemente en una variable del tipo estructura).

Si queremos aumentar nuestro mensaje con un dato entero llamado `selector`, incorporamos a la estructura un par *tipo-dato*:

```
struct
{
   mach_msg_header_t Cabecera;

   mach_msg_type_t Selector_tipo;
   mach_msg_type_t Selector_dato;
}
```

Podemos añadir tantos pares *tipo-dato* de parámetros a enviar al destinatario como queramos.

Ya tenemos nuestro mensaje de petición estructurado: démosle ahora un nombre apropiado, como un tipo C:

```
typedef struct
{
   mach_msg_header_t Cabecera;

   mach_msg_type_t Selector_tipo;
   mach_msg_type_t Selector_dato;
} peticion_t;
```

El siguiente paso es instanciar una variable de tipo `peticion_t`

```
peticion_t peticion;
```

y, por supuesto, montar los valores apropiados en la variable. Comencemos por la definición de tipo del entero llamado `selector` (véase la estructura de `mach_msg_type_t` en el apartado anterior):

```
peticion.Selector_tipo.msgt_name = 2;           /* ??? */
peticion.Selector_tipo.msgt_size = 32;          /* Es un entero de 32 bits */
peticion.Selector_tipo.msgt_number = 1;         /* ??? */
peticion.Selector_tipo.msgt_inline = TRUE;      /* Parece razonable para tan solo un
                                                   entero */
peticion.Selector_tipo.msgt_longform = FALSE;   /* No hace falta un formato largo */
```

```
peticion.Selector_tipo.msgt_deallocate = FALSE;  /* Nosotros mismos nos ocuparemos de
                                                    liberar esta memoria */
peticion.Selector_tipo.msgt_unused = 0;          /* Como su nombre indica */
```

Evidentemente, el dato se copia sin mas:

```
peticion.Selector_dato = selector;
```

Donde *selector* es el valor que queremos enviar en el parámetro entero.

Ahora debemos montar la cabecera del mensaje. Nada mas simple, conociendo la estructura de una cabecera:

```
peticion.Cabecera.msgh_bits = MACH_MSGH_BITS(numbits,
                                             tipo-mensaje);

peticion.Cabecera.msgh_size = sizeof(peticion_t);
peticion.Cabecera.msgh_request_port = puerto de destino
peticion.Cabecera.msgh_reply_port = mach_get_reply_port();
peticion.Cabecera.msgh_seqno = 0;
peticion.Cabecera.msgh_id = algún id
```

Obsérvese que pedimos al kernel un puerto de réplica para utilizar en este mensaje. Tanto el número de secuencia como el *id* del mensaje dependen de cómo los interprete la tarea destinataria (recordemos que estamos haciendo IPC a mano. Si los mensaje los montara MIG, utilizaría `id` para especificar el *MIG ID* de la rutina de servicio que debe aceptar este mensaje).

Finalmente, my ufanos, enviamos nuestro bonito mensaje a alguna incauta tarea utilizando, como no, la operación IPC `mach_msg`:

```
resultado = mach_msg (&peticion.Cabecera,
                      MACH_SEND_MSG|MACH_MSG_OPTION_NONE,
                      32,
                      &peticion.Cabecera.msgh_size,
                      peticion.Cabecera.msgh_reply_port,
                      MACH_MSG_TIMEOUT_NONE,
                      MACH_PORT_NULL);
```

... y tan contentos.

7.3 Ficheros

En *gnu-osfmk*, la implementación del sistema de IPC se encuentra en 'gnu-osfmk/ipc'. A continuación se listan los ficheros que componen la implementación y una breve descripción de su contenido.

'ipc_types.h'
: Este fichero de cabecera contiene las definiciones de dos tipos básicos, en concreto de *ipc_space_t y *ipc_port_t

'port.h'
: Define tipos relacionados con puertos como mach_port_index_t y mach_port_gen_t.

'ipc_entry.c ipc_entry.h'
'ipc_hash.c ipc_hash.h'
'ipc_init.c ipc_init.h'
'ipc_kmsg.c ipc_kmsg.h'
'mqueue.c mqueue.h'
'ipc_notify.c ipc_notify.h'
'ipc_object.c ipc_object.h'
'ipc_port.c ipc_port.h'
'ipc_pset.c ipc_pset.h'
'ipc_right.c ipc_right.h'
'ipc_space.c ipc_space.h'
'ipc_splay.c ipc_splay.h'
'ipc_table.c ipc_table.h'
: Gestión (creación, destrucción, etc) de tablas de capacidades IPC.

'ipc_thread.c ipc_thread.h'
'mach_debug.c'
'mach_msg.c mach_msg.h'
'mach_port.c'
'mig_log.c'
'ipc_print.h'
'ipc_machdep.h'

7.4 Espacios IPC

Cada una de las tareas del sistema dispone de un *espacio* de capacidades IPC. Este espacio es utilizado por las operaciones IPC como enviar y recibir mensajes. Las llamadas IPC al kernel manipulan el espacio de la tarea correspondiente.

7.4.1 Estructura ipc_space

```
struct ipc_space {
        decl_mutex_data(,is_ref_lock_data)
        ipc_space_refs_t is_references;

        decl_mutex_data(,is_lock_data)
        boolean_t is_active;            /* is the space alive? */
```

```
        boolean_t is_growing;           /* is the space growing? */
        ipc_entry_t is_table;           /* an array of entries */
        ipc_entry_num_t is_table_size;  /* current size of table */
        struct ipc_table_size *is_table_next; /* info for larger table */
        struct ipc_splay_tree is_tree;  /* a splay tree of entries */
        ipc_entry_num_t is_tree_total;  /* number of entries in the tree */
        ipc_entry_num_t is_tree_small;  /* # of small entries in the tree */
        ipc_entry_num_t is_tree_hash;   /* # of hashed entries in the tree */
        boolean_t is_fast;              /* for is_fast_space() */
};
```

7.4.2 Creación de espacios IPC

7.4.3 Creación de espacios IPC especiales

7.4.4 Destruyendo espacios IPC

7.4.5 Obteniendo una referencia en un espacio

7.4.6 Dejando una referencia en un espacio

7.5 Inicializando el sistema de IPC

7.5.1 ipc_bootstrap

Parte de la inicialización que debe hacerse antes de que sea creada la *tarea kernel*.

1. Se crean los espacios de puertos
2. Otras inicializaciones:
   - `mig_init();`
   - `ipc_table_init();`
   - `ipc_notify_init();`
   - `ipc_hash_init();`
   - `ikm_cache_init();`

7.5.2 ipc_init

Apéndice A Copiando Este Libro

Apéndice B GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant

Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgments", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this

License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

B.0.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

(No existe el Índice)