Raspberry Pi

Ignacio Javier Kovacs

April 2014

Índice

1. Introducción

En este proyecto se realizo un pequeño sistema operativo para Raspberry Pi, con el objectivo de introducirse en la arquitectura ARM.

El texto se encuentra dividido en 2 partes. La primera es una comparación de la arquitectura IA-32 de Intel contra ARM. Se describen las diferencias más relevantes observadas durante la implementación del kernel del sistema, y se da una descripción de las características de ciertas partes de la arquitectura.

La segunda parte habla sobre la implementación de determinadas partes del código del sistema.

Finalmente se presenta un instructivo para poder construir y probar el sistema en cuestión. Lo necesario para construir el sistema se adjunta en un dvd.

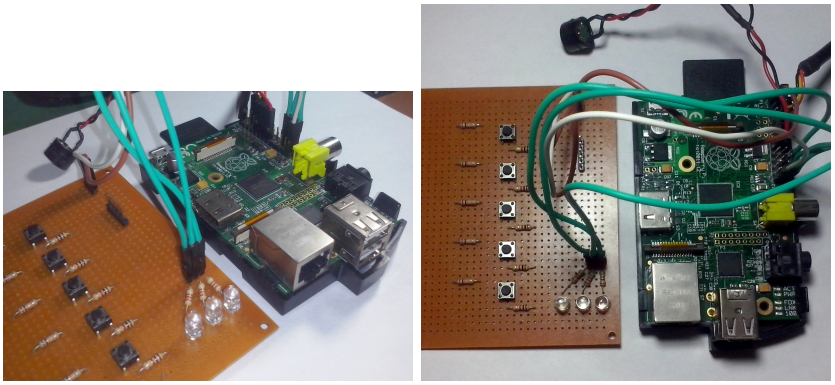


Figura 1: Hardware utilizado en el proyecto.

2. Raspberry Pi

2.1. Acerca de la Raspberry Pi

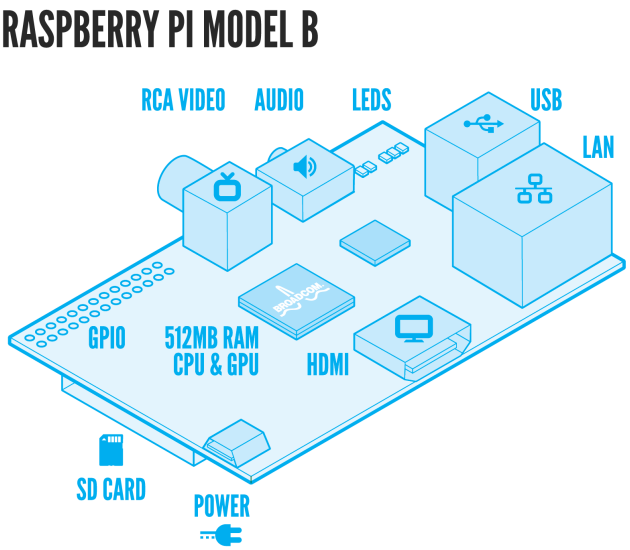


Figura 2: Raspberry Pi Modelo B.

La Raspberry Pi es lo que se denomina *SoC* o *System on Chip*. Es básicamente una GPU, CPU y memoria en un sólo chip.

Esta versión de la Raspberry Pi (Modelo B, rev. 2), cuenta con lo siguiente.

1. CPU ARM11 ARM1176JZF-S
2. GPU Broadcom VideoCore IV
3. RAM 512 Megabytes
4. Salida de audio (3.5mm jack).
5. Salida de video compuesto RCA.
6. Salida de video HDMI.
7. 2 Puertos USB 2.0
8. 1 Puerto 10/100Mb Ethernet.
9. 1 Secure Digital SD/MMC/SDIO card slot.
10. Pines de propósito general, UART.

2.2. Secuencia de Arranque

La Raspberry Pi no inicia como una computadora convencional. Es el procesador gráfico (GPU) quién inicia antes que el procesador ARM (CPU).

Al principio la GPU se encuentra encendida y la CPU no. La GPU ejecuta una serie de gestores de arranque (en etapas) hasta finalmente llegar al punto de entrada del kernel.

El primer gestor de arranque se encuentra en la memoria ROM del dispositivo. Este se encarga de cargar el segundo gestor de arranque. Para lograrlo debe acceder a la memoria externa del dispositivo y buscar el archivo “bootcode.bin”. Por último debe depositar el gestor en la cache asociativa L2.

El segundo gestor de arranque carga un tercer archivo, esta vez en memoria RAM, luego de haberla habilitado. Este archivo, llamado “loader.bin”, es el tercer gestor de arranque.

El tercer gestor de arranque lee y carga el firmware de la GPU, el cuál se encuentra en el archivo “start.elf”. Este último es un mini sistema operativo propietario que se encarga de levantar la imagen de nuestro kernel y realizar diferentes configuraciones. Al terminar, se lanza la señal de reset del CPU e inicia la ejecución de nuestro kernel.

3. ARM

3.1. Instrucciones

ARM utiliza un conjunto de intrucciones reducido o *RISC* (*Reduced Instruction Set*). Este hecho lo hace notablemente diferente a los procesadores Intel IA-32 en varios aspectos.

Las intrucciones utilizadas en ARM son de 32-bits de longitud. No es posible ingresar direcciones de tamaño arbitrario como operando debido a esta característica. Las direcciones y constantes de 32-bits deben ser cargadas en un registro previamente. El siguiente ejemplo muestra un fragmento de código que evidencia lo expuesto.

```
ARM_assembly:
ldr r0, =_32bit_address_  @ r0 := Dirección etiqueta _32_bit_address_
ldr r1, [r0] @ r1 := 0xf0000000
...

Intel_assembly:
mov eax, [_32_bit_address_] @ eax := 0xf0000000
...

_32_bit_address_:
.word 0xf0000000
```

Dentro del conjunto de instrucciones de ARM, muchas de las mismas poseen 3 operandos. El siguiente extracto ejemplifica como afecta esto, por ejemplo, a la operación de suma de registros en ambas arquitecturas. En el caso de ARM, el operando destino puede ser un tercer registro.

```
ARM_add:
add r3, r0, r1  @ Resultado r3 := r0 + r1
...

Intel_add:
add eax, ebx  @ Resultado eax := eax' + ebx
...
```

Las instrucciones en ARM no pueden operar directamente en memoria. La arquitectura utiliza instrucciones especiales de carga (*Load*) y almacenado (*Store*). Veamos la diferencia que esto implíca, por ejemplo, al tener que sumar 2 números de 32-bits de memoria y luego almacenarlos.

```
_numero1_:
.word 3
_numero2_:
.word 4

Intel_assembly:
mov eax, [_numero2_] @ eax := 4
add [_numero1_], eax @ Almacenar en _numero1_ el resultado de 4 + 3
...

ARM_assembly:
```

```
ldr r0, =_numero2_  @ r0 := _numero2_
ldr r1, [r0]  @ r1 := 4
ldr r0, =_numero1_  @ r0 := _numero1_
ldr r2, [r0]  @ r2 := 3
add r1, r1, r2  @ r1 := 4 + 3
stm r1, [r0]  @ Almacenar en _numero1_ r1
...
```

En ARM, los 2 números son previamente almacenados en registros, al igual que las direcciones de los mismos en memoria. Intel no tiene esa dificultad ya que se puede operar en memoria directamente, y el operando fuente o destino (no ambos) puede ser una constante de 32 bits de longitud.

Las operaciones *ldr* y *str* pueden direccionar a memoria de las siguientes formas.

Addressing mode and index method	Addressing syntax
Preindex with immediate offset	[Rn, #+/-offset_12]
Preindex with register offset	[Rn, +/-Rm]
Preindex with scaled register offset	[Rn, +/-Rm, shift #shift_imm]
Preindex writeback with immediate offset	[Rn, #+/-offset_12]!
Preindex writeback with register offset	[Rn, +/-Rm]!
Preindex writeback with scaled register offset	[Rn, +/-Rm, shift #shift_imm]!
Immediate postindexed	[Rn], #+/-offset_12
Register postindex	[Rn], +/-Rm
Scaled register postindex	[Rn], +/-Rm, shift #shift_imm

Las operaciones que toman *base + desplazamiento*, se encuentran limitadas por el tamaño de las instrucciones. Sólo se pueden ingresar desplazamientos de 12 bits.

Los modos de indirección utilizados por la instrucción *mov* en Intel, pueden ser simulados con lo descripto anteriormente. Por ejemplo, las siguientes 2 rutinas tienen el mismo efecto.

```
ARM_rutina:
@ r2 := indice
ldr r1, =dir_tabla
ldr r0, [r1, r10, LSL #2]

Intel_rutina:
@ ebx := indice
mov eax, [ebx * 4, dir_tabla]
```

Intel posee instrucciones específicas para realizar operaciones de entrada/salida a dispositivos, como *inb* y *outb*. ARM no dispone de un equivalente para estas instrucciones. Como los dispositivos son asignados a memoria (*Memory Mapped*), las mismas instrucciones de carga y almacenado son utilizadas para acceder a ellos.

Las llamadas a subrutinas en ARM no dejan la dirección de retorno en la pila. Existe un registro especial llamado *lr* o *link register*, en el cuál la dirección de retorno es almacenada cada vez que se ejecuta una instrucción de *branch & link*. Las intrucción de *brach & link* es equivalente a la instrucción *call* utilizada por Intel. Si se realizan sucesivas llamadas, el registro debe ser resguardado para evitar perder las direcciones de retorno.

```
ARM_add:
bl ARM_subrutina  @ lr := dirección de retorno.
...

ARM_subrutina:
stmfd sp!, {lr} @ Se preserva la dirección de retorno anterior. Equivalente a 'PUSH lr'.
bl otra_subrutina  @ lr := Dirección de retorno a ARM_subrutina.
...

Intel_add:
call Intel_subrutina  @ Se almacena en la pila la dirección de retorno.
...

Intel_subrutina:
call otra_subrutina  @ Se almacena en la pila la dirección de retorno.
...
```

Otra cuestión es que el procesador ARM no cuenta con un módulo de división entera. Las operaciones de división deben ser implementadas, utilizando código. Las divisiones por 0 deben ser manejadas por software, la arquitectura no posee una excepción para este fin.

3.1.1. Modos de Operación

Además del modo habitual de operación, ARM cuenta con 2 tecnologías *Jazelle* y *Thumb*.

Jazelle
: Los procesadores que poseen esta arquitectura permiten ejecución de Java bytecode de forma nativa en el hardware.

Thumb
: Es un subconjunto de instrucciones del modo ARM que tienen 16 bits de longitud. Tiene mayor desempeño en procesadores con un bus de datos de 16 bits de ancho. Además los programas escritos con estas instrucciones tienen una mayor densidad de código.

La arquitectura IA-32 no posee ninguna de estas 2 características.

3.2. Modos de Ejecución

La arquitectura IA-32 cuenta con varios modos de ejecución. Se utilizan más que nada para preservar la compatibilidad hacia atrás entre las distintas arquitecturas de Intel. Algunos de ellos son:

Real Mode
: Los procesadores IA-32 inician en este modo. Utiliza el modelo de segmentación. Las direcciones son de 20-bits de longitud máximo.

Protected Mode
: Este modo soporta memoria virtual que posee mecanismos de protección. Se utilizan direcciones de 32-bits, o 36-bits utilizando PAE (*Physical Address Extension*). Es compatible hacia atrás con el modo real (*Real Mode*).

Virtual 8086 Mode
: Provee la habilidad de ejecutar código de 16-bits, dentro del modo protegido (*Protected Mode*).

En oposición, ARM cuenta con 7 modos de ejecución. El propósito de los modos de ejecución difiere totalmente con los de Intel. La siguiente es una breve descripción de los distintos modos exitentes en ARM.

- Privilegiados

  Abort
  El procesador entra en este modo cuando falla al intentar acceder a una posición de memoria.

  Interrupt Request (IRQ) y Fast Interrupt Request (FIQ)
  Estos modos corresponden con 2 niveles de interrupción disponibles en el procesador.

  Supervisor
  Modo en el cual la CPU incia, luego del reset. Es el utilizado para ejecutar el kernel de sistema.

  System
  Este modo es una versión especial del modo usuario (user mode), sólo que privilegiado.

  Undefined
  El procesador utiliza este modo cuando encuentra una instrucción que no es soportada.

- No Privilegiados

  User
  Es el único modo no privilegiado. Es utilizado por programas y aplicaciones.

El modo activo define los registros utilizados y los privilegios de acceso sobre el registro *CPSR* (*Current Program Status Register*). El *CPSR* se utiliza para monitorear y controlar operaciones internas del procesador. Este registro cumple una función similar a el registro *EFLAGS* en Intel.

Cada modo en ARM puede ser clasificado como *privilegiado* o *no privilegiado*. A diferencia de la arquitectura IA-32, ARM no dispone de anillos de protección. Sin embargo se podría hacer un paralelo entre los anillos de protección de nivel 0 (kernel de sistema) y 3 (usuario), y esta clasificación entre modos.

Ejecutar dentro de un modo privilegiado permite leer y escribir en cualquier sección del registro *CPSR*. Las tareas de administración de sistema (como por ejemplo activar o desactivar la mmu) sólo pueden hacerse dentro un modo con privilegios.

Por otro lado un modo no privilegiado no permite operaciones de escritura sobre el *CPSR*, con excepción de los flags de programa.

El cambio de un modo a otro puede realizarse explícitamente escribiendo en la sección de control del registro *CPSR*, siempre y cuando se ejecute dentro de un modo privilegiado. También se produce de forma automática durante una excepción o interrupción.

3.3. Registros

Anteriormente mencionamos que el modo de ejecución define los registros activos. El procesador cuenta con 37 registros en total.

No todos los registros se encuentran activos en todos los modos, 20 de ellos son *banked registers*. Los mismos son reemplazados por otros al cambiar de modo.

Esta característica permite que el cambio de contexto entre modos de ejecución sea más rápido. Al igual que en la arquitectura IA-32 se utilizan diferentes pilas para cada nivel de protección, en ARM sucede lo mismo pero para cada modo. Durante un cambio de contexto de un modo a otro, el hecho de que la dirección de la pila utilizada se encuentre en un *banked register*, hace que no sea necesario realizar operaciones a memoria.

En la figura 3 podemos ver el conjunto de registros completo según el modo de operación. En la mayoría de los modos, el procesador cuenta con 18 registros: 16 de propósito general y 2 de estado del procesador.

Los registros de propósito general se identifican con la letra *r* seguido de un número. Todos los registros son de 32-bits de longitud y pueden contener datos o direcciones.

Algunos de estos registros son asignados a tareas específicas, a ser:

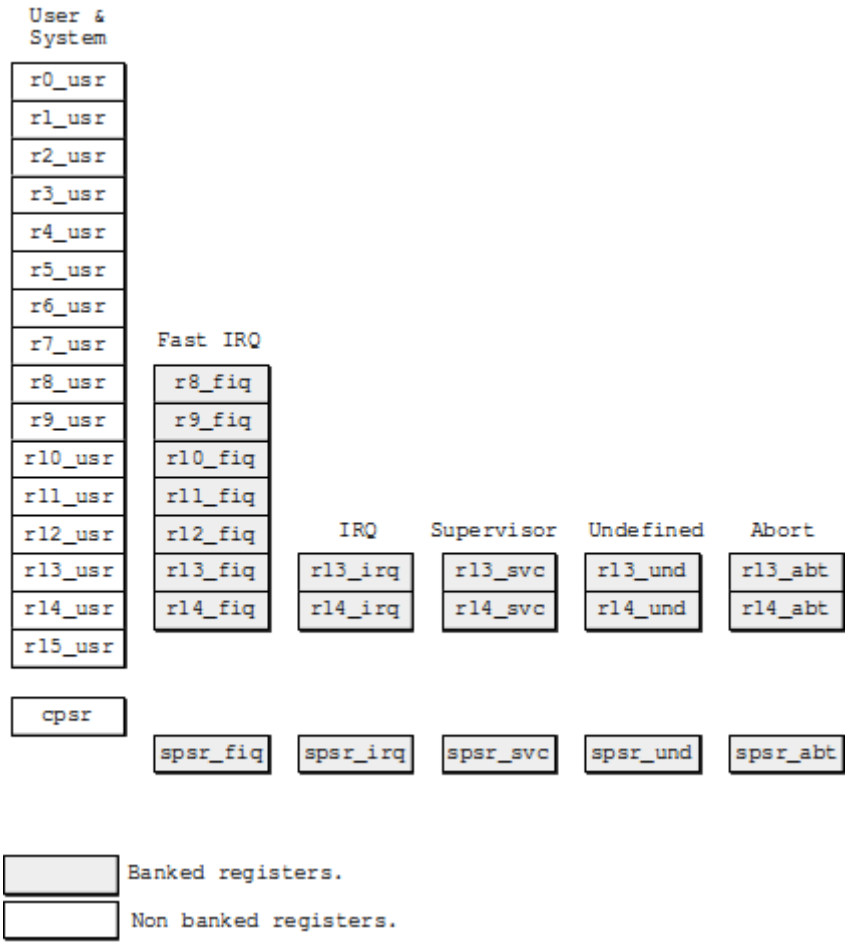


Figura 3: Registros del procesador.

Registro R13 o *lr*
Contiene la dirección de retorno de una subrutina.

Registro R14 o *sp*
Este registro es utilizado como puntero a pila.

Registro R15 o *pc*
Almacena la dirección de la siguiente instrucción a ser ejecutada.

El procesador cuenta con 2 registros de estado.

CPSR* o *Current Process Status Register Se utiliza para monitorear y control operaciones internas del procesador. El mismo tiene 32-bits de longitud y se divide en 4 secciones: *flags*, *status*, *extension* y *control*. El contenido de cada sección varia según el modelo de procesador. [1]

SPSR* o *Saved Process Status Register Este registro no se encuentra disponible en todos los modos y su utilización se hará clara más adelante. Basta con decir que el procesador lo utiliza para resguardar el estado del registro *CPSR*.

El modo usuario y sistema comparten el mismo set de registros, sólo que un modo es privilegiado y el otro no. Ambos modos, además, no poseen registro *SPSR*.

En los demás modos los registros desde *R0* hasta *R12* son los mismos, los únicos que cambian son aquellos marcados como *banked registers*.

El modo *Fast Interrupt Request* posee 5 *banked registers* adicionales, para reducir el impacto del cambio de contexto durante una interrupción.

[1] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/I2837.html Descripción detallada CPSR del procesador ARM1176JZF-S.

3.4. Excepciones

ARM no hace una distinción entre una interrupción y una excepción. El término es empleado indistintamente para referirse a una u otra cosa.

El manejo de excepciones en ARM es muy diferente al de la arquitectura IA-32. No existe una tabla de descriptores asociados a las rutinas de atención de interrupción. Cada excepción/interrupción se encuentra ligada a una entrada de una tabla conocida como *vector table*. Esta tabla se encuentra en la dirección *0x00000000* en memoria. En algunos procesadores ARM se tiene la posibilidad de elegir ubicarla en la dirección *0xffff0000*.

Cada entrada de la tabla contiene una intrucción de 32-bits de longitud. La misma modifica el contador de programa para hacer referencia al inicio de una rutina de atención. Las instrucciones utilizadas para este fin pueden ser varias: *b* (branch), *ldr* (load) o *mov* (mov).

En contraste con las 256 interrupciones disponibles en IA-32, ARM sólo posee 7. En Intel se tiene la posibilidad de fijar un nivel de privilegio para cada interrupción, en el descriptor de la misma. ARM define para cada interrupción un modo de ejecución predeterminado.

Las excepciones o interrupciones disponibles son las listadas a continuación, entre paréntesis el modo asociado.

Reset (Supervisor)
: Esta interrupción se produce ni bien se enciende el procesador. De aquí se salta al código de incialización del kernel, es el punto de entrada.

Undefined Instruction (Undefined)
: Cuando el procesador no puede decodificar una instrucción se produce esta excepción. Similar a la excepción *Invalid Opcode* en Intel.

Software Interrupt (Supervisor)
: Utilizada por las aplicaciones de usuario para realizar llamadas a sistema.

Data Abort (Abort)
: Sucede cuando una instrucción intenta acceder a memoria sin los permisos adequados. También cuando la dirección virtual especificada no está disponible. Su contrapartida en Intel sería la excepción *Page Fault*.

Prefetch Abort (Abort)
: Se aplica el mismo comportamiento que para la excepción *Data Abort*, sólo que el error refiere a la obtención de una instrucción y no un dato.

Interrupt Request (IRQ)
: Es utilizado por el hardware externo para interrumpir la ejecución del procesador. Sólo se produce de estar habilitadas las interrupciones (indicado por bit en *CPSR*, sección control).

Fast Interrupt Request (FIQ)
: Es similar a Interrupt Request. Es utilizado cuando se requiere mayor velocidad de respuesta. Posee la ventaja de tener un mayor número de *banked registers*.

Cuando sucede una interrupción el procesador:

1. Almacena la dirección de retorno en R14 del modo asociado a la excepción.
2. Copia el CPSR en el SPSR del modo asociado a la excepción.
3. Cambia los bits del CPSR para forzar el cambio de modo.
4. Obtiene y ejecuta la instrucción almacenada en la entrada de la tabla de excepciones para dicha excepción.

Las excepciones poseen niveles de prioridad. Puede darse la posibilidad de que se produzcan 2 excepciones al mismo tiempo, por lo que se le da prioridad a aquella de mayor nivel.

También las interrupciones modifican los bits asociados a las interrupciones externas del registro CPSR. En todos los casos al producirse una excepción se deshabilitan las interrupciones normales (no las rápidas).

La siguiente tabla contiene las prioridades y valores para los bits *I* (Interrupt Request) y *F* (Fast Interrupt Request) del CPSR. Un valor de 1 para cualquiera de los 2 bits, deshabilita las interrupciones del tipo en cuestión. Un guión significa que no es modificado.

Exceptions	Priority	I bit	F bit
Reset	1	1	1
Data Abort	2	1	-
Fast Interrupt Request	3	1	1
Interrupt Request	1	1	-
Prefetch Abort	4	1	-
Software Interrupt	5	1	-
Undefined Instruction	6	1	-

Cuando una excepción ocurre, el link register es configurado a una dirección específica en función del contador de programa. Por ejemplo cuando sucede un Data Abort, el link register contiene la dirección de la última instrucción ejecutada más 4. Nosotros debemos ajustar el link register de forma que apunte a la instrucción que generó el error. Como la instrucción que generó el error fue la última ejecutada, debemos restar 8 (recordar que 1 instrucción tiene 4 bytes) al registro lr para volver a ejecutarla una vez corregido el error.

La siguiente tabla define las correcciones que deben realizarse durante la excepción al link register para retornar a la intrucción correcta.

Exception	Return address	Use
Reset	-	lr is not defined on a Reset
Data Abort	lr - 8	points to the instruction that caused the Data Abort exception
FIQ	lr - 4	return address from the FIQ handler
IRQ	lr - 4	return address from the IRQ handler
Prefetch Abort	lr - 4	points to the instruction that caused the Prefetch Abort exception
SWI	lr	points to the next instruction after the SWI instruction
Undefined Instruction	lr	points to the next instruction after the undefined instruction

3.5. Administrador de memoria virtual y protección (ARMv6)

En la arquitectura IA-32 es necesario configurar segmentación para iniciar la mmu, por la compatibilidad hacia atrás se implementa el mecanismo de memoria virtual sobre segmentación. En ARM no es necesario ya que no utiliza segmentación.

Al igual que Intel, ARM utiliza tablas de páginas para realizar la traducción de direcciones. Cada entrada de la tabla permite otorgar a cada página permisos para los distintos niveles de privilegio, además de políticas de cache y buffer de escritura.

ARMv6 utiliza un directorio de páginas de 2 niveles. La tabla de primer nivel soporta páginas de 1M (*section*) y 16M (*supersection*), conocida como *master* o *level 1 page table*. Esta posee 4096 entradas de 4 bytes, cada una hace referencia a una *section* (sección) que direcciona 1M de memoria. Cuando se utilizan páginas de 16M, la entrada se replica 16 veces en la tabla a partir de la *section* de inicio.

La tabla de segundo nivel, o *coarse table*, tiene sólo 256 entradas de 4 bytes cada una. Soporta páginas de 4K (*small pages*) y 64K (*large pages*). Al igual que con las *supersections*, las entradas asociadas a páginas de 64K se repiten 16 veces.

Tradicionalmente a diferencia de ARM, la arquitectura IA-32 sólo soporta 2 tamaños de páginas de 4K y 4M. ARM no cuenta con *PAE* o *Physical Address Extension*, sólo puede manejar un espacio de memoria virtual de 4G.

Sistemas operativos como Linux dividen el espacio de direcciones virtuales en 2, una para las aplicaciones de usuario y otra para el kernel de sistema. Devido a esto, ARM tiene la habilidad de partir su directorio de páginas en 2. De esta forma el espacio asociado a la aplicación de usuario puede ser reemplazado sin necesidad de invalidar todo el directorio. El procesador posee 2 registros que almacenan las direcciones base de ambas tablas *translation table base 0 (ttb0) y 1 (ttb1)*. Esta característica es opcional, y se encuentra desactivada por defecto. De no utilizarse, sólo se realiza la traducción con la tabla que indique el registro *ttb0*.

3.5.1. Dominios

Cada sección de la tabla de nivel 1 tiene asociado un dominio, existen 16 dominios configurables en ARMv6. El acceso a memoria se ve perjudicado dependiendo del tipo de dominio seleccionado. Los tipos de dominio son los siguientes.

Client
: Clientes son los usuarios de los dominios en donde se ejecutan programas y guardan datos. Cada acceso al dominio debe ser verificado contra los permisos de la página correspondiente.

Manager
: Los administradores controlan el comportamiento del dominio, las secciones y páginas del mismo y los derechos de acceso. Por este motivo las secciones bajo este dominio no son protegidas, no se verifican los permisos.

3.5.2. Permisos de acceso

Cada página contiene permisos de acceso para ambos niveles de privilegio. Si una página es accedida sin los permisos necesarios se produce una excepción Data Abort o Prefetch Abort, dependiendo del contexto. La siguiente tabla muestra las posibles configuraciones de permisos permitidas.

APX	AP	Supervisor	User
0	b00	No access.	No access.
0	b01	Read/write.	No access.
0	b10	Read/write.	Read-only.
0	b11	Read/write.	Read/write.
1	b00	Reserved.	Reserved.
1	b01	Read-only.	No access.
1	b10	Read-only.	Read-only.
1	b11	Read-only.	Read-only.

Otros atributos configurables son:

Shared
: Indica que el área de memoria puede ser compartido por múltiples procesadores.

Never Execute
: Habilita la ejecución de código en la página.

Not-Global
: Este bit indica si la traducción es global, o específica de un proceso en la TLB. De ser específica de un proceso, la traducción ingresa en la TLB relacionada con un *ASID* (*Address Space Identifier*). Ver sección Translation Lookaside Buffer debajo.

3.5.3. Políticas de caché

Los bits *C* y *B* están relacionados con el comportamiento de la caché y el buffer de escritura respectivamente. Las siguientes configuraciones son aceptadas.

C & B bits	Policy
b00	Noncacheable
b01	Write-Back cached, Write Allocate
b10	Write-Through cached, No Allocate on Write
b11	Write-Back cached, No Allocate on Write

3.5.4. Translation Lookaside Buffer

ARM posee 2 niveles de *TLB* o *Translation Lookaside Buffer*. La TLB de primer nivel es la *microTLB*, es más pequeña y de mayor velocidad que la principal. Existen microTLB separadas para los datos y las instrucciones, ambas poseen 10 líneas de tipo *fully associative*.

La TLB de segundo nivel o principal, actúa cuando se produce un *miss* en alguna de las microTLB (instrucciones o datos). Al contrario de la microTLB el procesador sólo posee 1 cache asociativa tanto para datos como para instrucciones. La TLB principal contiene 8 líneas *fully associative* y otra región que consta de 64 líneas *low associative*.

La arquitectura soporta *Address Space Identifier* o ASID, para permitir al sistema diferenciar entre los espacios de direcciones utilizados por los procesos. De esta manera no es necesario vaciar toda la cache durante un cambio de contexto.

Las entradas también pueden ser marcadas como *global*, de esta forma son compartidas por los espacios virtuales de todos los procesos.

Aunque los tipos de caches empleados por Intel no son iguales, se utiliza el mismo sitema de 2 niveles (Intel Core i7). También dispone de su propia implementación de ASID (llamdo PCID o Process Context Identifier) y entradas globales.

4. Sistema Operativo

4.1. Interrupciones

Las excepciones Undefined Instruction, Data & Prefetch Abort no cumplen ninguna función de peso en este sistema operativo. Son utilizadas como medio para obtener información acerca de los problemas que puedan surgir. Por ese motivo la única tarea que realizan es avisar, dar información del error y detener la ejecución del sistema. Las rutinas de atención de las 3 excepciones son implementadas de forma similar, todas muestran un volcado de los registros al momento de la interrupción. Cada interrupción, además, brinda información extra a ser la dirección en donde se generó y el motivo.

Por ejemplo, el siguiente código en la tarea *init* ocasiona un error en los permisos de escritura.

```
int main()
{
    *((uint32_t *) 0) = 4;
    ...
```

La tarea no tiene privilegios de escritura en la dirección *0x00000000*, por lo que se produce una excepción. La excepción se imprime junto con la causa y un detalle del contenido de la pila al momento de la interrupción.

```
[ex] Data Abort
[db] 0x00000000 - Fault Address Register
[db] Data Fault Register
        @ AXI decode error caused the abort
        @ Write access caused the abort
        @ Status: Permission section fault

[db] Stack
        0x7FFFFFBC: 0x00000000 (0)
        0x7FFFFFC0: 0x00000000 (0)
        0x7FFFFFC4: 0x00000004 (4)
        0x7FFFFFC8: 0x00000000 (0)
        0x7FFFFFCC: 0x00000000 (0)
        0x7FFFFFD0: 0x00000000 (0)
        0x7FFFFFD4: 0x00000000 (0)
        0x7FFFFFD8: 0x00000000 (0)
        0x7FFFFFDC: 0x00000000 (0)
        0x7FFFFFE0: 0x00000000 (0)
        0x7FFFFFE4: 0x00000000 (0)
        0x7FFFFFE8: 0x7FFFCFFC (2147471356)
        0x7FFFFFEC: 0x00000000 (0)
        0x7FFFFFF0: 0x7FFFFFF8 (2147483640)
        0x7FFFFFF4: 0x40000000 (1073741824)
        0x7FFFFFF8: 0x40000014 (1073741844)
        0x7FFFFFFC: 0x00000050 (80)

[db] System halt
```

La rutina de atención de la excepción Interrupt Request se utiliza para averiguar la procedencia de la misma y luego atenderla. Los dos dipositivos que utilizan esta interrupción son el reloj de sistema, y el puerto de comunicación serial mini UART al recibir o enviar datos.

Las llamadas a sistema se realizan a través de la excepción *Software Interrupt* (*SWI*). La misma se disparará cuando un programa o aplicación ejecuta la instrucción *swi*. La intrucción toma un número de 24-bits que utiliza para saber de que llamada a sistema se trata. Durante la ejecución de la rutina, se obtiene la instrucción swi ejecutada para obtener este número. El número indexa una tabla de direcciones donde se encuentran las funciones de sistema. Luego realiza una llamada a la función pedida, de forma transparente.

La excepción Fast Interrupt Request no es utilizada.

4.2. Administración de memoria física

La memoria RAM del dispositivo se administra mediante un mapa de bits. Cada bit en el mapa representa una sección de la misma de tamaño igual a 4K.

El mapa de bits está divido en dos partes, la parte con las direcciones más bajas (área de sistema) se utiliza para crear las estructuras que necesita el sistema. Esta área se encuentra virtualizada utilizando *identity mapping*. De esta forma no debemos virtualizar una página previamente a utilizarla como estructura para el sistema de paginación.

El área de direcciones más altas (área usuario) se usa para contener código y datos. Esta región no se encuentra virtualizada.

Luego de la creación del mapa de bits, se alojan las páginas ya utilizadas: el área donde reside el kernel, la pila, el vector de interrupciones, etc. Las direcciones de dichas secciones se obtienen de las variables definidas en el proceso de enlazado.

Se brindan funciones para las siguientes operaciones.

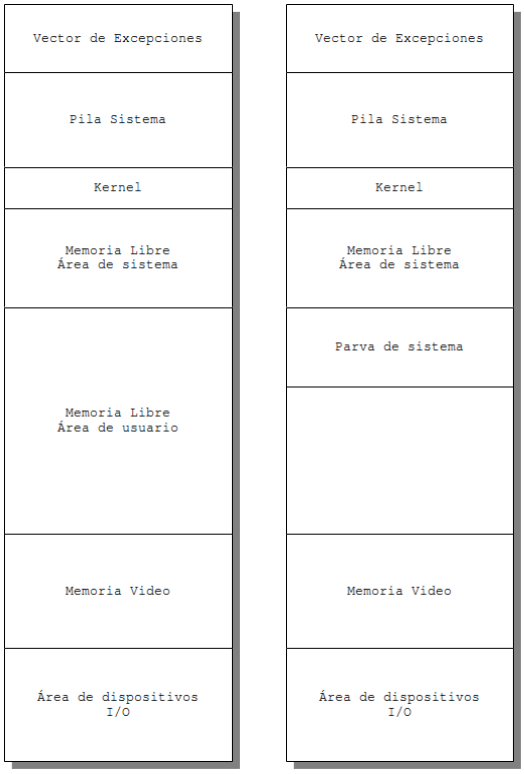
1. Pedir 1 página de 4K.
2. Pedir 1 página de 4K, previamente virtualizada. Utilizadas para generar las tablas de traducción de nivel 2 (Coarse).
3. Pedir 4 páginas de 4K contiguas, previamente virtualizadas. Se utiliza más que nada para generar las tablas de traducción de nivel 1 (Master).
4. Liberar páginas no utilizadas.

4.2.1. Organización de la Memoria

La figura 4 muestra la disposición de la memoria física, y como la misma se encuentra virtualizada.

El vector de excepciones, la pila utilizada por el sistema, el kernel, dispositivos de I/O, memoria libre de sistema y de video, se encuentran virtualizadas con *identity mapping* (es decir, sus direcciones físicas y virtuales son las mismas).

En el espacio de direcciones virtuales el heap del kernel se ubica donde comienza la memoria de usuario libre, a continuación del área de sistema.



(a) Mapa de memoria física. (b) Mapa de memoria virtual.

Figura 4: Memoria virtual y física.

4.3. Heap del sistema

El administrador de memoria virtual se encarga de buscar bloques de memoria libres para su uso. Se implementa utilizando listas doblemente enlazadas, una de bloques libres y otra de bloques usados.

Cada bloque de memoria pedido al sistema posee un encabezado que contiene un par de punteros indicando el próximo bloque libre (en caso de estar libre) o usado (en caso de estar en uso). Además posee información sobre el tamaño de dicho bloque.

El administrador también guarda la ubicación de los comienzos de ambas listas y de su tamaño total, en la primera sección del heap.

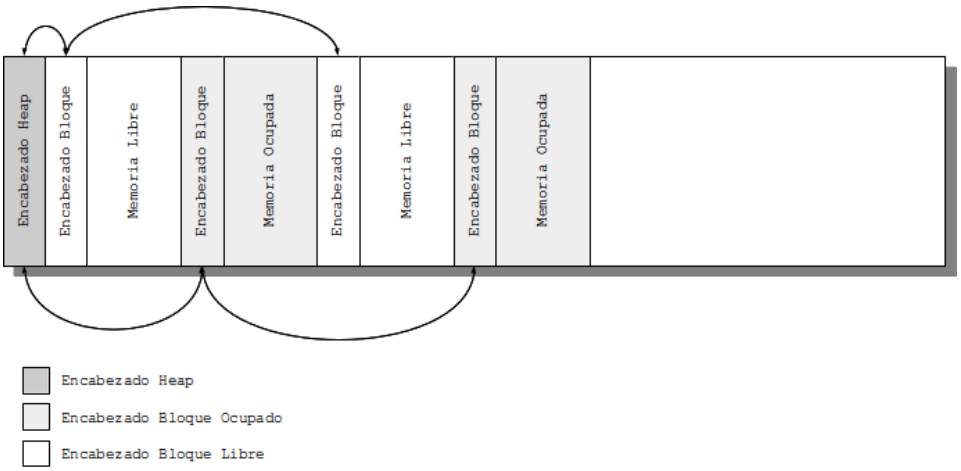


Figura 5: Estructura generada por el administrador de memoria.

Cada vez que se desea un bloque de memoria se busca el bloque libre más grande (Worst Fit Policy). A continuación se lo secciona en 2 partes formando 2 bloques, de manera que el bloque de memoria restante

quede en la lista y no deba ser vuelto a insertar. El otro bloque se inserta al principio de la lista de usados. Finalmente se retorna la dirección del bloque de memoria listo para su uso. Si el bloque es del mismo tamño que el pedido, no es modificado y se retorna para ser utilizado.

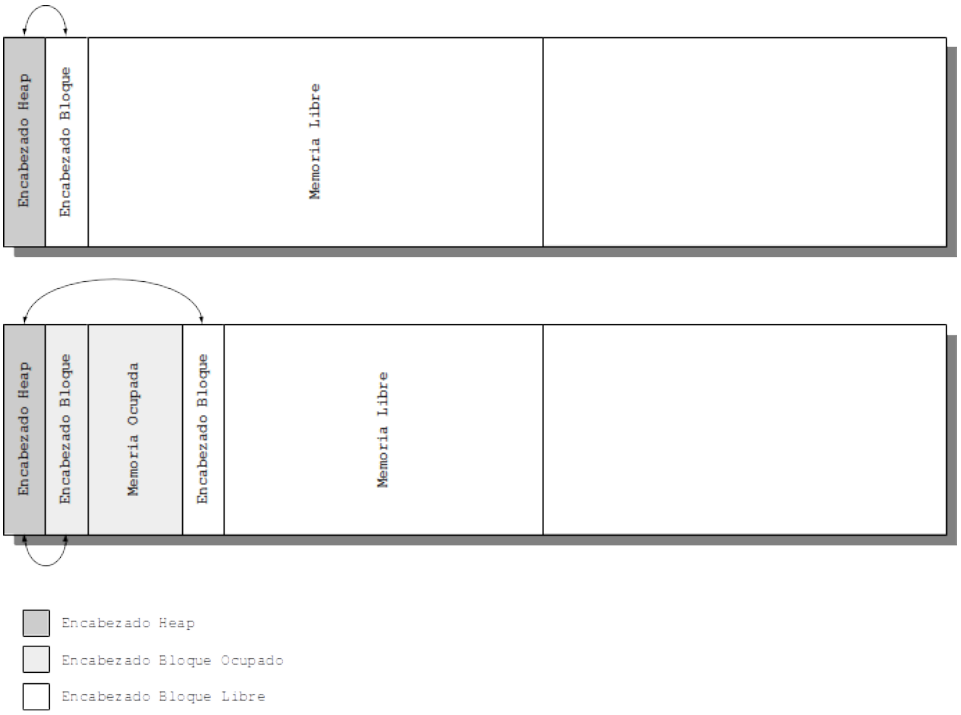


Figura 6: *malloc*, pedido de memoria.

Para devolver memoria, el sistema recupera la dirección del encabezado del bloque. Luego lo quita de la lista de usados, lo inserta por dirección en la lista de nodos libres y comprueba si puede funsionar el bloque con otro (también libre). Debido a que la lista de bloques libres se encuentra organizada por dirección es más fácil realizar la fusión de 2 bloques. Sólo debemos ver si el bloque siguiente y anterior son contiguos.

4.4. Procesos

Las tareas se compilan a parte y sus archivos en formato ejecutable (Elf 32 para ARM) se añaden a la sección de datos de sólo lectura del kernel. Al momento de cargar la tarea, el archivo Elf32 es analizado, y con la información obtenida se generan las secciones de la tarea.

Cada sección es copiada a una página y mapeada a la correspondiente dirección virtual, en la tabla de páginas del proceso.

Las tareas no solo son lanzadas con este sistema, también se utilizan las funciones *fork* y *exec* combinadas para clonar y reemplazar procesos, creando nuevas tareas.

Cada proceso se identifica con un PID (Process ID).

```
struct process_t
{
        int pid; // Identificador de proceso

        process_t *parent; // Puntero a proceso padre
        Queue *children; // Procesos hijos

        pcb_t *pcb; // Bloque de control de proceso

        uint32_t quantum; // Quantum (valor asignado)
        uint32_t ticks; // Ticks faltantes (contador usado por scheduler)
        uint32_t state; // Estado de proceso

        segment_t *text; // Segmentos de programa
        segment_t *rodata;
        segment_t *data;
        segment_t *bss;
        segment_t *svc_stack;
        segment_t *usr_stack;
        segment_t *heap;

        List *mutexes; // Mutex pedidos por el proceso
        segment_t *shared; // Memoria compartida, actualmente solo se puede
            compartir 1 pag de 4k
};
```

Figura 7: Estructura de un proceso.

La figura 8 muestra como está conformado el espacio virtual de los procesos. El mismo comienza en la dirección virtual *0x40000000*.

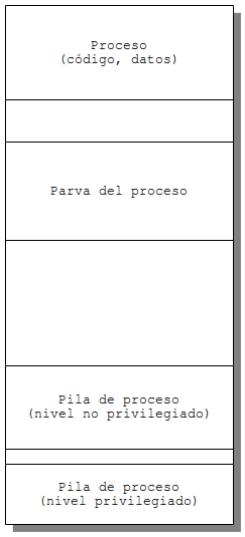


Figura 8: Imagen de un proceso en memoria virtual.

4.4.1. Clonar un proceso: *fork()*

fork() es una función utilizada para clonar un proceso. La misma devuelve el pid (Process Id) del proceso clonado.

En la estructura de proceso, se guarda información acerca del comienzo, longitud y attributos de cada sección de programa. Cada sección del programa se copia a una página física, la cual se virtualiza en un nuevo directorio de páginas. Este nuevo directorio de páginas se utilizará para el proceso clon. La única sección que no se replica es la memoria compartida, pero sí se registra en el directorio del nuevo proceso.

El último paso es inicializar la estructura *pcb_t* de tal forma que el nuevo proceso reanude la ejecución como si fuera el proceso original. Para ello se utiliza una función en ensamblador, *saveContext*.

Luego de salvar el contexto, los dos procesos quedan en el mismo estado y con los mismos datos. Lo que distingue entre uno y otro, y determina el valor de retorno de la función *fork()* es el siguiente código.

```
if(scheduler.running->pid != copy->pid)
{
        // Si soy quien ejecuto fork ...
        // Ingreso la tarea clon en el planificador.
        scheduler_admit(copy);

        // Devuelvo el pid de la nueva tarea.
        return copy->pid;
}

// Si soy la copia... devuelvo 0.
return 0;
```

Si lo vemos desde la perspectiva de la tarea que ejecutó la función *fork()* originalmente, debe devolver el pid del proceso generado. También debe ubicar la tarea en la cola del planificador para su posterior ejecución.

Cuando vemos que sucede en el proceso clonado, la ejecución se retoma al volver de la función *fork()*. El valor devuelto es 0 ya que se trata del proceso clonado.

4.4.2. Reemplazar un proceso: *exec()*

La función *exec()* modifica el espacio virtual de un proceso para convertirlo en otro. Las secciones (*segment_t**) del proceso original son descartadas y reemplazadas por las secciones del nuevo.

A pesar de deshechar las secciones de código y datos, muchas otras se conservan. Las dos pilas (nivel privilegiado y no privilegiado), se restauran. El heap de la aplicación libera toda la memoria que fue tomada por el proceso anterior y es reutilizado. La sección compartida no se deshecha, es heredada por la nueva tarea.

La estructura *pcb_t* es reinicializada y se utiliza la función *loadContext* para comenzar la ejecución.

4.5. Planificador de Tareas

Como planificador de tareas se implemento un Virtual Round Robin. El planificador tiene 2 colas, una la llamaremos “ready” y la otra “auxiliary” (ver figura 9).

En un Round Robin convencional cada tarea se ejecuta por una cantidad de tiempo finito. Cuando se le acaba el tiempo, el planificador desaloja la tarea y le da lugar a otra. En este sistema también sucede lo mismo. Cada vez que una tarea gasta su tiempo de ejecución, llamado quantum, es desalojada y encolada en “ready”. Para cargar una nueva tarea el planificador desencola una tarea de la cola “ready” o “auxiliary”, dependiendo de cuál no esté vacía. La cola “auxiliary” tiene mayor prioridad que la otra, es decir que el planificador intentará primero quitar una tarea de esta.

Una tarea puede bloquearse por una operación de entrada o salida a un dispositivo o simplemente ceder el control de ejecución en determinado momento (poner la tarea a dormir). Al hacerlo, la tarea no gasta la totalidad de su quantum y queda bloqueada. Una tarea bloqueada no reingresa a las colas del planificador debido a que no volverá a ejecutarse hasta que se complete la operación de entrada o salida. Cuando la tarea se desbloquee, esta ingresara en la cola “auxiliary” y el valor de quantum será aquel que no gastó a causa del bloqueo. Cuando el bloqueo termina la tarea está lista para procesar, es por eso que la cola “auxiliary” tiene más prioridad.

Las casusas por las cuales la tarea puede bloquearse son por el uso de las funciones sleep, getch, putch y la utilización de variables candado.

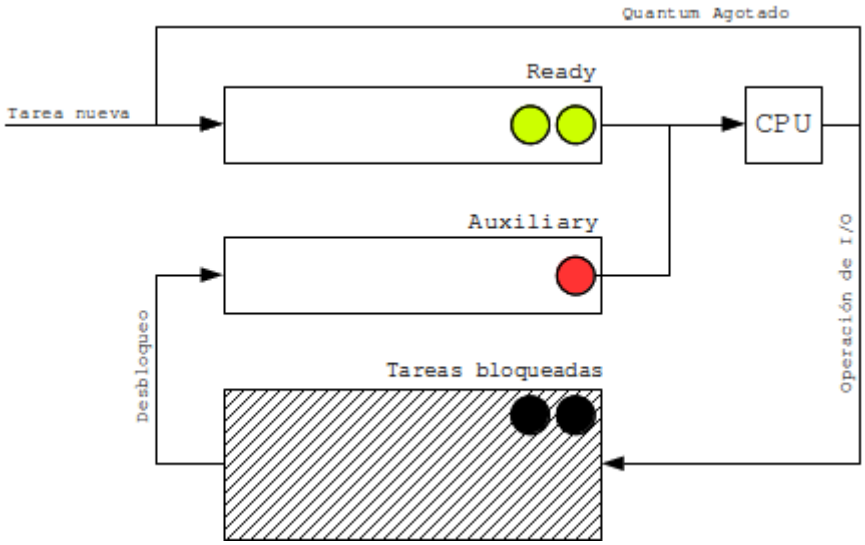


Figura 9: Virtual Round Robin.

Gracias a el reloj de sistema, el cuál interrumpe cada 1 milisegundo, se puede saber cuando una tarea a terminado su turno en la CPU. Cada tarea es ejecutada por 50 milisegundos (quantum). Como valor mínimo para amortizar los tiempos muertos, se utiliza un valor de 20 milisegundos. Es decir, si la tarea se bloqueó y su quantum restante es de 5 milisegundos, este se incrementa a 20 automáticamente.

4.6. Cambio de contexto

El código del listado 10, es la rutina utilizada para cambiar de contexto. La misma almacena todos los registros utilizados por la tarea en cuentión en una estructura conocida como *pcb_t* (Process Control Block).

Para poder acceder a los registros del modo usuario, es necesario cambiar al modo Sistema. En el mismo se tiene acceso a los registros de modo usuario pero sin perder los privilegios. Se realiza de esta forma porque es imposible regresar a un modo privilegiado luego de haber ingresado al modo usuario, debido a que no se tiene permiso de escritura necesario sobre el registro *CPSR*.

La tabla de traducciones de páginas es reemplazada por la del próximo proceso. La *TLB* es vaciada para prevenir errores en la traducción (borrar las traducciones del proceso anterior).

Una vez terminado el cambio de contexto, la instrucción *movs* restaura el contador de programa y el estado previo del registro *CPSR*.

```
contextSwitch:
        @ r0 := Current Task Control Block
        @ r1 := Next Task Control Block

        stmfd sp!, {r1} @ Almacenar r1
        cps #Sys32md
        stmea r0, {r0-r14} @ Guardar R0-R14
        cps #SVC32md
        mov r1, sp
        add r1, #4
        str r1, [r0, #60] @ Guardar sp_svc corregido, +4 (debido al primer
           stmfd)
        str lr, [r0, #64] @ Guardar lr_svc (retorno a rutina)
        mrs r1, cpsr
        str r1, [r0, #68] @ Guardar spsr_svc
        mrc p15, 0, r1, c2, c0, 0
        str r1, [r0, #72] @ Guardar ttb0
        ldmfd sp!, {r0} @ Cargar tcb tarea siguiente
        ldr r1, [r0, #72]
        mcr p15, 0, r1, c2, c0, 0 @ Cargar ttb0
        mov r1, #0
        mcr p15, 0, r1, c8, c7, 0 @ Flush TLB
        ldr r1, [r0, #68]
        msr spsr, r1 @ Restaurar spsr
        ldr r1, [r0, #64]
        mov lr, r1 @ Restaurar lr
        ldr r1, [r0, #60]
        mov sp, r1 @ Restaurar sp
        cps #Sys32md
        ldmfd r0, {r0-r14} @ Restaurar registros de usuario
        cps #SVC32md
        movs pc, lr @ Volver
```

Figura 10: Cambio de contexto entre 2 procesos.

4.6.1. Espera no bloqueante: *sleep()*

La función *sleep* se utiliza como espera no activa. La tarea es puesta a dormir y por lo tanto retirada de la CPU durante una cantidad fija de milisegundos (pasados a la función). Una vez que el plazo de tiempo acordado se cumplió, el sistema vuelve a colocar el proceso en la cola de mayor prioridad.

Llamemos a las tareas TX, con X un número natural (T1, T2, etc). Además notaremos t(TX) al tiempo que la tarea debe de esperar.

Para saber cuando una tarea a finalizado su espera se utiliza una cola delta (delta queue). Las tareas ingresan a la cola con un valor igual a t(TX). Por ejemplo en la figura 11, se ve ingresar a la delta queue a la tarea 1 (T1) con t(T1) igual a 10 (milisegundos).

Figura 11: La tarea 1 (T1) ingresa a la cola con $t(T1) = 10$.

El valor t(TX) del primer elemento se verá reducido en 1 unidad luego de cada milisegundo (marcado por el reloj de sistema). Cuanto t(TX) llegue a cero se desencola la tarea y desbloquea.

Si ingresara una segunda tarea (T2) se compara t(T1) con t(T2). En caso de t(T2) ser mayor a t(T1), ver figura 12, se inserta T2 detrás de T1 con valor $t'(T2) = t(T2) - t(T1)$. De esta forma cuando se quite a T1 de la cola, faltarán $t(T2) - t(T1)$ milisegundos para despertar a T2.

Figura 12: La tarea 2 (T2) ingresa a la cola con valor 15.

Si t(T2) fuera menor que t(T1), T2 se posicionaría delante de T1 con su valor t(T2) sin modificar. El valor de T1 se vería reducido en t(T2), $t'(T1) = t(T1) - t(T2)$.

Figura 13: La tarea 2 (T2) ingresa a la cola con valor 5.

4.6.2. Variables candado: *Mutex*

El sistema utiliza las interrupciones como sistema de exclusión mutua. Esto puede hacerse gracias a que las interrupciones no son reentrantes.

Se cuenta con 3 llamadas a sistema que posibilitan su utilización.

1. *mutex_create*, crea un mutex.
2. *mutex_adquire*, pide la variable candado.
3. *mutex_release*, libera el mutex.

Cuando se crea un mutex, el sistema lo anexa a un vector. Es la posición en ese vector lo que se utiliza como "key" o "id" para identificar la variable candado.

Cuando se realiza una llamada a sistema, se desactivan las interrupciones. Por esta razón el planificador de tareas no desalojará a la tarea actual. Se utiliza este comportamiento como sistema de exclusión al momento de leer y escribir la variable candado, ya que no habrá ningún otro proceso ejecutándose.

Cuando el mutex es solicitado, de estar "bloqueado", la tarea que lo pidió es desalojada. El proceso permanecerá bloqueado hasta que otro libere el recurso. La tarea desalojada es encolada en una estructura utilizada sólo por el mutex en cuestión.

Al momento de liberarse un mutex, si la cola asociada tenía procesos esperando, se desencolará uno e inmediatamente será puesto a ejecutarse.

La figura 14 muestra como es el proceso de bloquear un mutex, mientras que la figura 15 muestra como se libera.

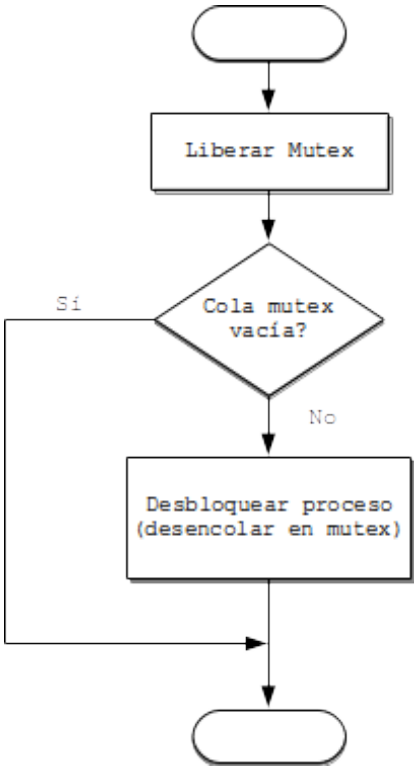


Figura 14: Liberar un mutex.

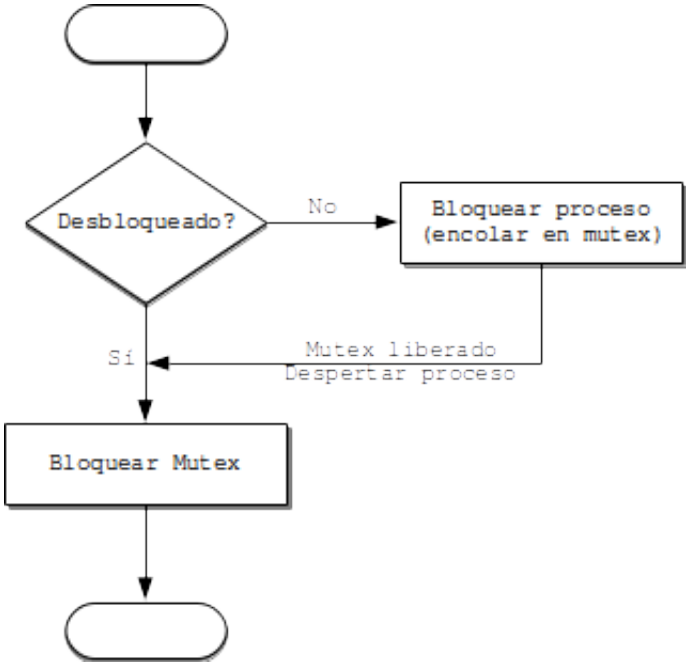


Figura 15: Adquirir un mutex.

4.6.3. Impresión por puerto serial: *print()*

La función *print()*, es el equivalente a *printf()* de la librería estándar de C. Internamente utiliza la función *putch()*, para imprimir de a un caracter a la vez.

Si dos tareas corriendo sobre el mismo procesador utilizaran la función print a la vez, un cambio de contexto podría producir que se mezclen las salidas de ambas. Es por eso que se creo un mutex (su ID es 0), para regular el acceso a la salida. Cada vez que se llama a la función en cuestión, la primer tarea a realizar es pedir este mutex para obtener el acceso exclusivo a la salida. Una vez que la impresión concluye, el mutex es liberado.

```
#define print(fmt, ...) \
mutex_adquire(0); \
__print(fmt, __VA_ARGS__); \
mutex_release(0)
```

En el código de arriba se puede apreciar la utilización del mutex.

4.7. Controlador Mini UART

La Raspberry Pi dispone de un puerto serial mini UART (implementación mínima de un PL011). Una rutina de incialización le configura de la siguiente forma:

- Sin control de errores
- Sin control de flujo
- Dato de 8-bits
- 1 bit de parada
- 9600 baudios

Se implemento un controlador para el mismo, de manera tal que pueda reemplazar la funcionalidad del teclado. Las funciones *getch* y *putch* se utilizan para obtener y enviar un caracter de 1 byte a través del puerto.

El sistema utiliza buffers circulares para almacenar los caracteres recibidos y a transmitir. Ambas funciones bloquean el proceso de no ser posible completar la acción satisfactoriamente.

4.7.1. *putch()*

La función *putch* intenta colocar un caracter en el buffer de transmisión. Si el buffer se encuentra lleno, la tarea que invocó a *putch* será bloqueada hasta que el buffer tenga al menos un lugar libre.

El puerto mini UART interrumpe cada vez que el buffer de salida físico tiene al menos 1 posición disponible. Cuando esto sucede, los bytes almacenados en el buffer circular de transmisión son tranportados al buffer del puerto. Finalmente si una tarea estaba esperando para escribir, es nuevamente activada.

El diagrama 16 muestra la operación descripta.

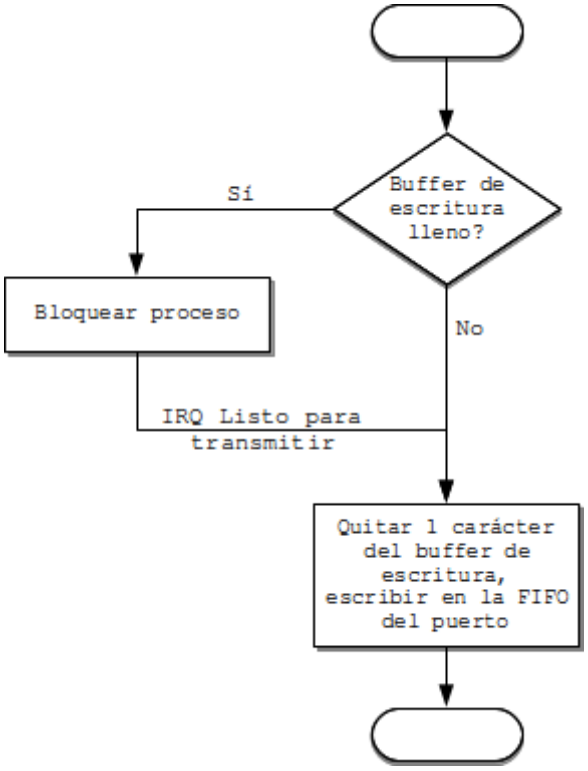


Figura 16: Llamada a sistema *putch*.

4.7.2. *getch()*

Posee la misma forma de proceder que *putch*. Esta vez se bloquea un tarea sólo cuando el buffer circular de recepción está vacío. Si el buffer de recepción está vacío, se bloquea la tarea hasta que un byte sea recibido.

Cuando se recibe un byte por el puerto, este ingresa en el buffer físico del mismo. También se produce una interrupción, en la misma se leen los datos recibidos y se vuelcan sobre el buffer circular de recepción. Además, si una tarea se encontraba esperando, se desbloquea.

A continuación un diagrama con la descripción de la operación 17.

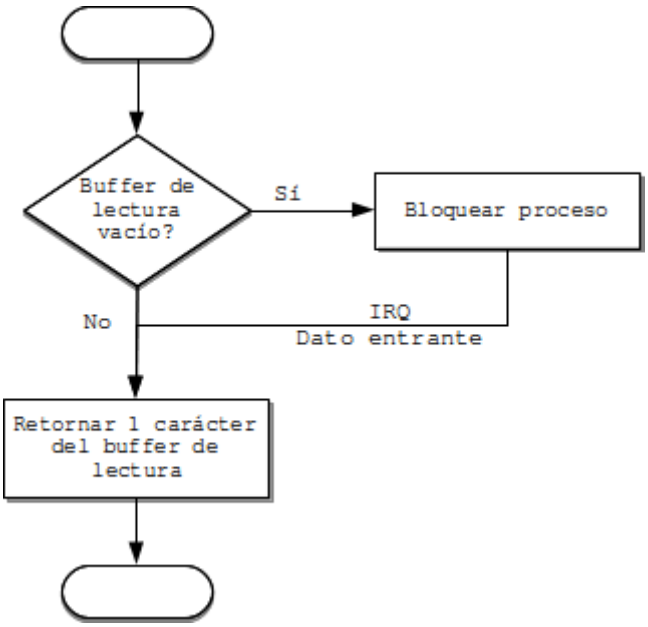


Figura 17: Llamada a sistema *getch*.

4.8. Pines de propósito general

El kernel posee 3 *syscalls* dedicadas al manejo de los pines de propósito general.

gpio_fsel
: La misma se utiliza para seleccionar la función del pin elegido. Puede ser entrada, salida o cumplir alguna otra función alternativa. Varios de los pines son utilizados por otros dispositivos (opcionalmente) como el UART, o el SPI.

gpio_set
: En caso de ser utilizado como salida, esta función envía una señal alta, interpretada como un 1 lógico (+3.3v), a través del pin especificado.

gpio_clear
: Operación inversa a *gpio_set*. Envía una señal baja, interpretada como un 0 lógico.

4.9. Tarea: init

Esta tarea es la primera en inciar. Una vez que se activan las interrupciones y el planificador empieza a funcionar (recibe interrupciones del reloj de sistema), el primer cambio de contexto se produce entre el sistema y la tarea init.

Init realiza una llamada a la función *fork*. Una copia del proceso asignado como hijo de init comenzará a ejecutarse luego de que las demás tareas en la cola del planificador terminen su quantum.

El hijo de init ejecutará *exec(1)*, convirtiendo así al proceso en la tarea speaker, de la cuál hablaremos en la siguiente sección.

Por otro lado, init, imprimirá un mensaje identificándose y manipulará un pin de propósito general para prender y apagar un led indefinidamente.

```
int main()
{
        int child = fork();

        if(child != 0)
        {
                gpio_fsel(25, FSEL_OUTPUT);

                print("Soy el proceso init, mi pid es %d\n", getpid());

                while(1)
                {
                        gpio_set(25);
                        sleep(300);
                        gpio_clear(25);
                        sleep(300);
                }

        }
        else
        {
                print("Soy hijo del proceso init, mi pid es %d\n", getpid());

                exec(1); // speaker
        }

        while(1);
        return 0;
}
```

Figura 18: Estracto de la tarea init.

4.10. Tarea: speaker

La función de la tarea será reproducir una canción utilizando un speaker (de PC).

La tarea speaker, al igual que el proceso init, utiliza *fork* para clonarse. El padre (la tarea speaker original) se encargará de simular una señal cuadrada de un período determinado por la nota que debe reproducirse. El hijo (el clon), deberá leer las notas de un arreglo y depositarlas en memoria compartida para que el proceso padre pueda saber que tocar. La duración de la nota está dada por la cantidad de tiempo que queda almacenada (sin cambiar) en memoria compartida. El tiempo de la duración de la nota se toma utilizando *sleep*.

La función *usleep* realiza espera activa, se utiliza de esta forma porque las resoluciones de tiempo que manejan son menores a 1 milisegundo que es lo mínimo que acepta *sleep*.

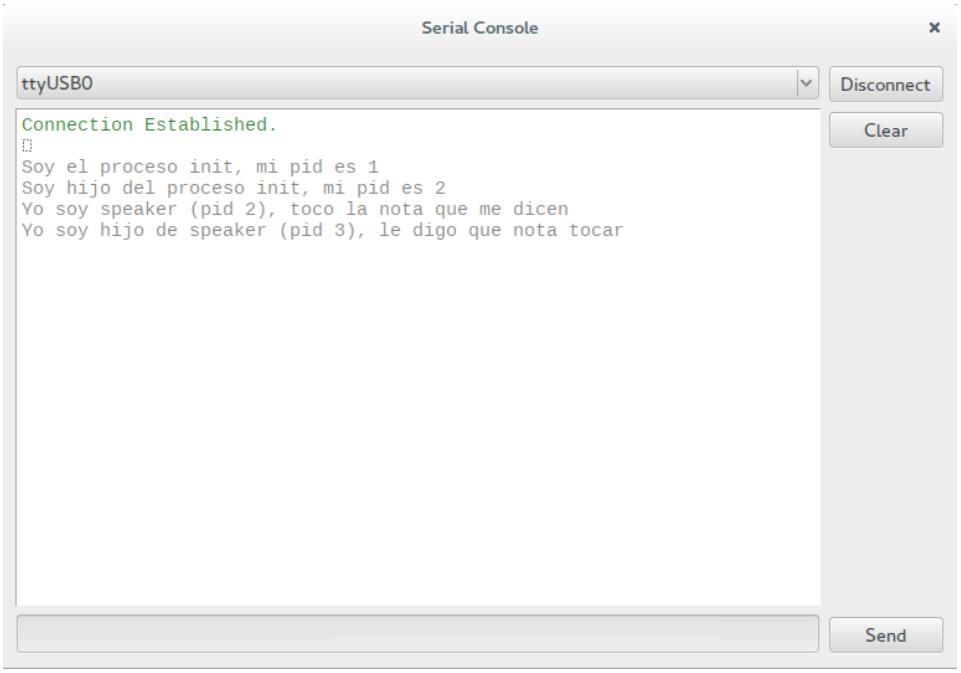


Figura 19: Salida por consola de la tarea speaker.

```
int main()
{
        gpio_fsel(pin, FSEL_OUTPUT);
        gpio_set(pin);

        int child;
        uint32_t *halfPeriod = share_mem();

        *halfPeriod = 0;

        child = fork();

        if(child != 0)
        {

                print("Yo soy speaker (pid %d), toco la nota que me dicen\n",
                    getpid());

                while(1)
                {
                        gpio_set(pin);
                        usleep(*halfPeriod);
                        gpio_clear(pin);
                        usleep(*halfPeriod);
                }
        }
        else
        {
                uint32_t note, duration, pause;

                print("Yo soy hijo de speaker (pid %d), le digo que nota
                   tocar\n", getpid());

                while(1)
                {
                        for(int i = 0; i < SONG02_SIZE; i++)
                        {
                                note = song02[0][i];
                                duration = 1000 / song02[1][i];
                                pause = (duration * 30) / 100;

                                tone(halfPeriod, note);
                                sleep(duration);

                                tone(halfPeriod, 0);
                                sleep(pause);
                        }
                }
        }
        return 0;
}
```

Figura 20: Estracto de la tarea speaker.

5. Instructivo

5.1. Paso 1: Obtener el hardware

Para este proyecto se utilizó una Raspberry Pi Modelo B, revisión 2.

Las tareas programadas interactúan con 3 LEDs y 1 speaker de PC. Los mismos fueron añadidos a la Raspberry como se muestra en la figura 21.

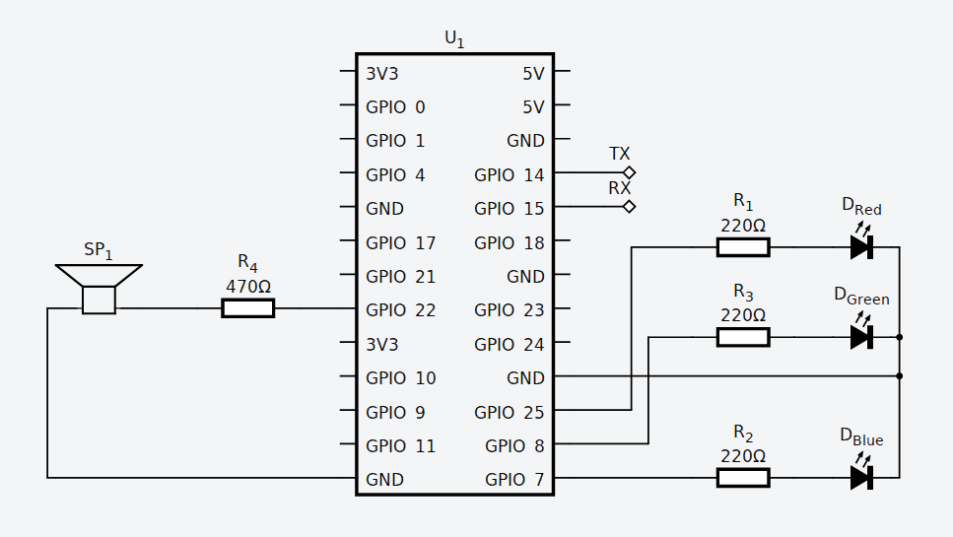


Figura 21: Raspberry Pi Modelo B, GPIO rev.2.

5.2. Paso 2: Preparar la tarjeta de memoria

Antes que nada, no todas las memorias SD son compatibles con la Raspberry Pi. Podemos ver un listado aquí http://elinux.org/RPi_SD_cards.

Lo primero será particionar, formatear y preparar nuestra tarjeta de memoria. Necesitaremos que la misma posea una partición W95 FAT32 (LBA) y dentro los siguientes archivos.

```
bootcode.bin    config.txt    fixup.dat              kernel.img     start.elf     cmdline.txt
fixup\_cd.dat   fixup\_x.dat  kernel\_emergency.img   start\_cd.elf  start\_x.elf
```

Por lo general las distribuciones de Linux compatibles con Raspberry, como Raspbian o Archlinux para ARM, se instalan mediante el uso de imágenes preparadas que pueden ser copiadas mediante alguna herramienta como *dd*.

A continuación veremos 2 métodos. El primer método, utiliza una imágen de una distribución de Linux para generar la partición que necesitamos y obtener los archivos mencionados. El segundo método, crea de forma manual la partición y utiliza los archivos enviados junto con el proyecto (*sdcard.tar.gz*).

5.2.1. Método 1

Si decidimos utilizar una distribución como Archilinux ARM (método utilizado en el proyecto), debemos bajar la imagen actualizada de la página http://archlinuxarm.org/platforms/armv6/raspberry-pi. Con la ayuda de alguna herramienta como *dd* debemos copiar la imagen descargada en la unidad de memoria.

```
[ignacio@ignacio-Desktop ~]$ sudo dd if=archlinux.img of=/dev/sdf bs=1M
```

Una vez finalizada la copia, debemos montar la partición FAT32 etiquetada *boot* en nuestro sistema de archivos.

```
Disk /dev/sdf: 1,9 GiB, 1990197248 bytes, 3887104 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x00047c7a

Disposit. Inicio Start     Final Blocks  Id System
/dev/sdf1         8192    122879  57344   c W95 FAT32 (LBA)
```

Dentro de la misma, como mínimo, deben existir los siguientes archivos mencionados.

Cuando construyamos la imágen del kernel de sistema, debemos de reemplazar el archivo *kernel.img* listado arriba por el nuestro.

5.2.2. Método 2

Para hacerlo utilizando el segundo método debemos insertar la tarjeta SD en la unidad de memoria e identificarle. Utilizaremos alguna herramienta de particionado para generar 1 sóla partición W95 FAT32 (LBA), y darle formato. A continuación se muestra un ejemplo utilizando el comando *fdisk* en Linux.

```
[ignacio@ignacio-Desktop ~]$ sudo fdisk /dev/sdf

Welcome to fdisk (util-linux 2.24.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Orden (m para obtener ayuda): o
Created a new DOS disklabel with disk identifier 0xe0fff63f.

Orden (m para obtener ayuda): n

Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Número de partición (1-4, default 1): 1
First sector (2048-3887103, default 2048): 2048
Last sector, +sectors or +size{K,M,G,T,P} (2048-3887103, default 3887103): 3887103

Created a new partition 1 of type 'Linux' and of size 1,9 GiB.

Orden (m para obtener ayuda): t
Selected partition 1
Hex code (type L to list all codes): c
If you have created or modified any DOS 6.x partitions,
 please see the fdisk documentation for additional information.
Changed type of partition 'Linux' to 'W95 FAT32 (LBA)'.

Orden (m para obtener ayuda): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
```

Para darle el formato se utilizó el siguiente comando.

```
[ignacio@ignacio-Desktop ~]$ sudo mkfs.vfat -F32 -n boot /dev/sdf1
mkfs.fat 3.0.26 (2014-03-07)
mkfs.fat: warning - lowercase labels might not work properly with DOS or Windows
```

Finalmente debemos montar la partición y copiar el comprimido *sdcard.tar.gz* (entregado aparte junto con los fuentes) y descomprimirlo.

```
[ignacio@ignacio-Desktop ~]$ sudo mount -t vfat /dev/sdf1 /mnt/flash/
[ignacio@ignacio-Desktop ~]$ cd /mnt/flash/
[ignacio@ignacio-Desktop flash]$ sudo cp ~/sdcard.tar.gz ./
[ignacio@ignacio-Desktop flash]$ sudo tar xzf sdcard.tar.gz
[ignacio@ignacio-Desktop flash]$ ls -al
total 28948
drwxr-xr-x 2 root root     4096 abr 11 09:50 .
drwxr-xr-x 3 root root     4096 abr 11 09:49 ..
-rwxr-xr-x 1 root root    17816 nov 17 03:00 bootcode.bin
-rwxr-xr-x 1 root root      142 nov 17 17:36 cmdline.txt
-rwxr-xr-x 1 root root     1234 mar  4 19:21 config.txt
-rwxr-xr-x 1 root root     2037 nov 17 03:00 fixup_cd.dat
-rwxr-xr-x 1 root root     5746 nov 17 03:00 fixup.dat
-rwxr-xr-x 1 root root     8779 nov 17 03:00 fixup_x.dat
-rwxr-xr-x 1 root root      137 jul 26  2013 issue.txt
-rwxr-xr-x 1 root root  9614632 nov 17 03:00 kernel_emergency.img
-rwxr-xr-x 1 root root   115028 abr 10 15:02 kernel.img
-rwxr-xr-x 1 root root 13389195 abr 11 09:50 sdcard.tar.gz
-rwxr-xr-x 1 root root   471256 nov 17 03:00 start_cd.elf
-rwxr-xr-x 1 root root  2497684 nov 17 03:00 start.elf
-rwxr-xr-x 1 root root  3476100 nov 17 03:00 start_x.elf
```

5.3. Paso 3: Construir el proyecto

Para construir el proyecto debemos descomprimir los fuentes. Se creara una carpeta titulada *ARM* con el siguiente contenido.

```
[ignacio@ignacio-Desktop ~]$ tar xzf ARM.tar.gz
[ignacio@ignacio-Desktop ~]$ ls ARM -al
total 15584
drwxr-xr-x  4 ignacio wheel     4096 abr 11 08:19 .
drwx------ 49 ignacio wheel     4096 abr 11 08:29 ..
-rw-r--r--  1 ignacio wheel     1185 abr 11 08:15 Makefile
-rwxr-xr-x  1 ignacio wheel 15937393 abr 11 08:23 Serial-RPi
drwxr-xr-x  2 ignacio wheel     4096 abr 11 08:14 src
drwxr-xr-x  3 ignacio wheel     4096 abr 11 08:17 tasks
```

Dentro del directorio *ARM* debemos descomprimir el toolchain de GNU para ARM (Sourcery CodeBench Lite 2013.05-23).

El mismo puede ser descargado desde
https://docs.google.com/file/d/0B8l8oXIW96f8bmNLOXptYXQ1bFk/edit.

Ya que el toolchain fue concebido para arquitecturas de 32-bits, puede que necesitemos instalar librerías adicionales. En el caso de Archilinux (x86_64), fue necesaria la instalación de glibc para 32-bits (paquete lib32-glibc).

Una vez descomprimida la carpeta, el directorio debe verse de la siguiente manera.

```
[ignacio@ignacio-Desktop ARM]$ ls -al
total 15588
drwxr-xr-x  5 ignacio wheel     4096 abr 11 08:49 .
drwx------ 49 ignacio wheel     4096 abr 11 08:29 ..
drwxr-xr-x  8 ignacio wheel     4096 ene 23 13:43 arm-none-eabi
-rw-r--r--  1 ignacio wheel     1185 abr 11 08:15 Makefile
-rwxr-xr-x  1 ignacio wheel 15937393 abr 11 08:23 Serial-RPi
drwxr-xr-x  2 ignacio wheel     4096 abr 11 08:14 src
drwxr-xr-x  3 ignacio wheel     4096 abr 11 08:17 tasks
```

A continuación sólo debemos ejecutar el comando *make*. Una vez terminada la construcción de la imagen del kernel, debemos realizar lo anteriormente mencionado en el paso 2, copiar el archivo *bin/kernel.img* a la partición *boot* reemplazando el existente (puede ser útil hacer una copia de seguridad primero del archivo kernel.img).

5.4. Paso 4: Encender la Raspberry Pi

Una vez concluídos todos los pasos anteriores, sólo resta colocar la tarjeta de memoria en la Raspberry Pi.

Si se dispone de un cable USB a TTL-RS232, podemos conectar el puerto mini UART de la Raspberry Pi al PC. En los fuentes se incluye un programa llamado *Serial-RPi*, que nos permite leer del puerto serial los mensajes enviados por el sistema y las aplicaciones. Debemos seleccionar el dispositivo y hacer click en *connect*, antes de encender la Raspberry Pi y después de conectar el cable del puerto.

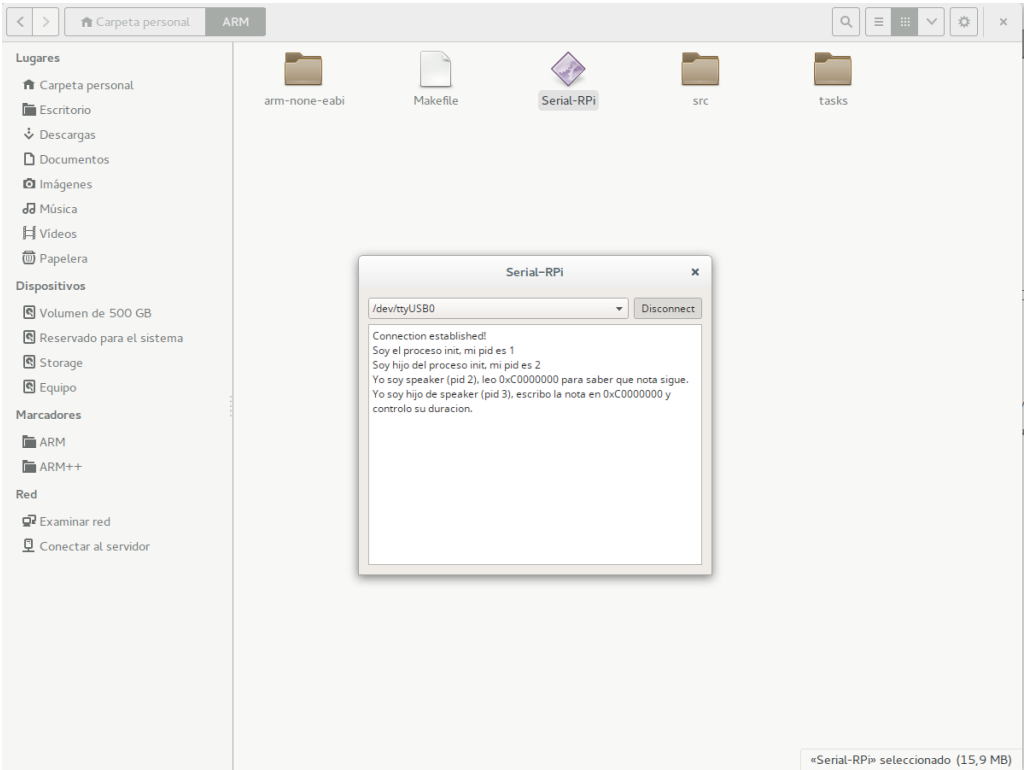


Figura 22: Serial-RPi

6. Material de Referencia

1. ARM1176JZF-S Technical Reference Manual
2. ARM - Architecture Reference Manual
3. BCM2835 ARM Peripherals
4. Executable and Linkable Format (ELF)
5. Procedure Call Standard for the ARM Architecture
6. ARM ELF File Format
7. Run-time ABI for the ARM Architecture
8. PrimeCell UART (PL011) Technical Reference Manual
9. ARM System Developers Guide - Designing and Optimizing System Software - Andrew N. Sloss, Dominic Symes & Chris Wright
10. http://www.raspberrypi.org/forums/
11. http://wiki.osdev.org/Expanded_Main_Page
12. http://elinux.org/RPi_Hub