μCronos: Arquitectura Básica de una Plataforma para Sistemas Móviles Extensibles

Victor D. Castillo Díaz, Rolando Menchaca Méndez.

Centro de Investigación en Computación-IPN
Unidad Profesional Adolfo López Mateos, C.P. 07738, México, D.F.
e-mail VictorCastilloEn@yahco.com.mx, rmen@cic.ipn.mx

RESUMEN

El presente artículo describe la arquitectura de μCronos, un micro núcleo para sistemas operativos dinámicamente extensibles. μCronos sirve como plataforma base para el desarrollo de proyectos relacionados con sistemas extensibles, cómputo ubicuo y sistemas móviles. μCronos tiene la capacidad de adaptarse dinámicamente a cambios del entorno, situación muy común en los dispositivos móviles y en general en los ambientes de cómputo ubicuo. Esto se logra principalmente mediante el uso de extensiones dinámicas sensibles al contexto de trabajo en el que se encuentre el dispositivo. Por otro lado, se analiza la arquitectura de procesadores ARM, ampliamente usada en sistemas portables y embebidos, para dejar ver sus diferencias con respecto a procesadores con arquitectura x86, sobre los cuales se desarrolla una primera implementación de la arquitectura. Esto con el fin de refinar detalles que nos permitan portar hacia arquitecturas móviles y embebidas basadas en ARM de forma confiable.

Palabras clave:
Arquitectura ARM, Capa Básica de Comunicaciones, Micro núcleo, Reconfiguración Dinámica, Sistemas Extensibles, Sistemas Móviles.

1. INTRODUCCIÓN

μCronos es un esfuerzo de desarrollo que contempla proveer un micro núcleo básico que dote a los sistemas móviles con la capacidad de adaptarse dinámicamente a los cambios de su entorno de operación.

El micro núcleo antes mencionado proporciona soporte básico para la administración de memoria, creación, configuración y destrucción de hilos, así como las primitivas necesarias para su sincronización, mecanismos de intercomunicación, interfaz para dispositivos de E/S. Además de soporte para la carga, configuración y descarga de objetos compartidos (módulos), que doten al sistema operativo de la funcionalidad requerida por sus subsistemas y las aplicaciones con que interactúa, todo esto de forma dinámica y bajo diferentes dominios de aplicación [1].

El objetivo de la implementación es proveer una base de software para la investigación y desarrollo de proyectos relacionados con sistemas operativos dinámicamente extensibles para sistemas móviles y sus diversas áreas de aplicación. Entre los objetivos de diseño de μCronos podemos mencionar:

- Estabilidad
- Interfaz para la Programación de Aplicaciones (API)
- Buen desempeño
- Soporte de Tiempo Real
- Soporte de administración de energía
- Soporte para extensiones dinámicas
- Una capa básica de comunicaciones (BCL)

Por otro lado, el proceso de desarrollo de μCronos se lleva a cabo bajo las siguientes prerrogativas:

1. Diseñar la arquitectura de una plataforma para sistemas móviles extensibles.
2. Implementar un micro núcleo para sistemas operativos dinámicamente extensibles utilizando el modelo de desarrollo estilo Bazar [2].
3. Diseñar e implementar componentes que se agreguen dinámicamente al micro núcleo y den soporte especifico a las necesidades del entorno de trabajo.
4. Utilizar el sitio web www.geocities.com/ViDaSoftwareInc/ como herramienta para la difusión y desarrollo cooperativo del proyecto.

El resto del artículo está organizado de la siguiente forma. En la sección 2, se describen las características de los ambientes y dispositivos en los que se ejecuta un micro núcleo del tipo de μCronos. En la sección 3, se describe a grandes rasgos la arquitectura de μCronos. En la sección 4

J. Díaz de León, G. González, J. Figueroa (Eds.): Avances en Ciencias de la Computación, pp. 340-350, 2003.

se describe, la capa básica de comunicaciones, que es la encargada de dar a µCronos la capacidad de hacer uso eficiente de los recursos restringidos de las redes inalámbricas. En la sección 5 se describe a detalle la arquitectura de µCronos, así como la de los módulos que lo componen. Finalmente, en la sección 6 se describen las conclusiones del presente trabajo.

2. ENTORNO DE OPERACIÓN

El objetivo de µCronos es servir como base para el desarrollo de aplicaciones ubicuas que se ejecuten en dispositivos móviles. El ambiente ubicuo impone restricciones tanto de hardware como de software que describiremos a continuación.

2.1 Hardware

Actualmente los equipos de cómputo han incrementado su capacidad de procesamiento y disminuido sus dimensiones de forma considerable, permitiendo el desarrollo de dispositivos muy compactos con un buen desempeño. Estos dispositivos generalmente son de aplicación especifica.

Una de las arquitecturas de hardware más comúnmente usadas en la construcción de dispositivos móviles y sistemas embebidos es ARM [3]. ARM se basa principalmente en arquitectura RISC aunque con algunas características CISC y provee bajo consumo de energía combinado con alto desempeño.

Las características principales de la arquitectura son [4]:

- Buena densidad de código
- Arquitectura RISC de 32 bits
- 16 registros de propósito general, incluido el contador de programa (PC)
- Soporte de memoria virtual
- Modos privilegiados
- Banco de registros

Como se aprecia en la Fig. 1, la arquitectura ARM cuenta con las interfaces necesarias para básicamente cualquier tipo de sistema.

La diferencia con respecto de otras arquitecturas es que está especializada en dispositivos móviles y embebidos, por lo tanto logra un buen desempeño a pesar de las restricciones de tamaño y consumo de energía.

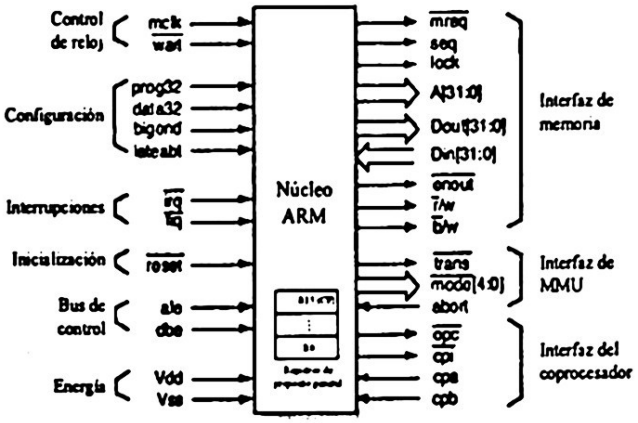

Figura 1. Interfaces del procesador basado en ARM.

2.2 Ambientes Ubicuos

En años recientes, una gran cantidad de investigadores se han dado a la tarea de materializar las ideas de Mark Weiser [5] y construir sistemas de cómputo ubicuo.

El desarrollo del computo ubicuo involucra varias ramas de las ciencias de la computación como el desarrollo de sistemas embebidos, la interacción hombre máquina, los sistemas sensibles al contexto, las redes de computadoras, los sistemas distribuidos y los sistemas extensibles.

En todas las líneas antes mencionadas el desarrollo de sistemas ubicuos impone nuevos desafíos y restricciones, por lo que muchas de las problemáticas son temas de investigación abiertos.

Un sistema ubicuo es en esencia un ambiente saturado con elementos de cómputo y comunicación, que están integrados con las tareas de sus usuarios humanos [6].

Tres de las características fundamentales de los sistemas ubicuos son la integración física entre elementos de la vida diaria y elementos de software, la interoperabilidad espontánea entre componentes ubicuos [7] y, la capacidad de percibir su contexto y tomar acciones en base a él.

Existen muchas razones por las cuales el diseño de sistemas ubicuos implica tratar con recursos computacionales limitados y dinámicamente variables. Como se mencionó, los sistemas embebidos tienden a ser pequeños, y la miniaturización significa recursos limitados.

En ese mismo orden de ideas, los recursos pueden variar considerablemente [7]. Un dispositivo en determinado ambiente puede tener acceso a una red inalámbrica de alto ancho de banda como la

IEEE 802.11b [8], y en otro puede tener acceso a una red de menor ancho banda como Bluetooth [9].

En un sistema ubicuo, los componentes deben interoperar espontáneamente en ambientes cambiantes. Un componente de este tipo cambia de compañeros durante su operación normal, conforme se mueve u otros componentes entran o salen de su entorno, ver Fig. 2.

3. ARQUITECTURA DE MICRO NÚCLEO DINÁMICAMENTE EXTENSIBLE

Como se muestra en la Fig. 3, la arquitectura micro núcleo implementa las abstracciones de bajo nivel del sistema operativo dentro del espacio del núcleo y permite que las abstracciones de alto nivel sean implementadas en el espacio del usuario [10,11].

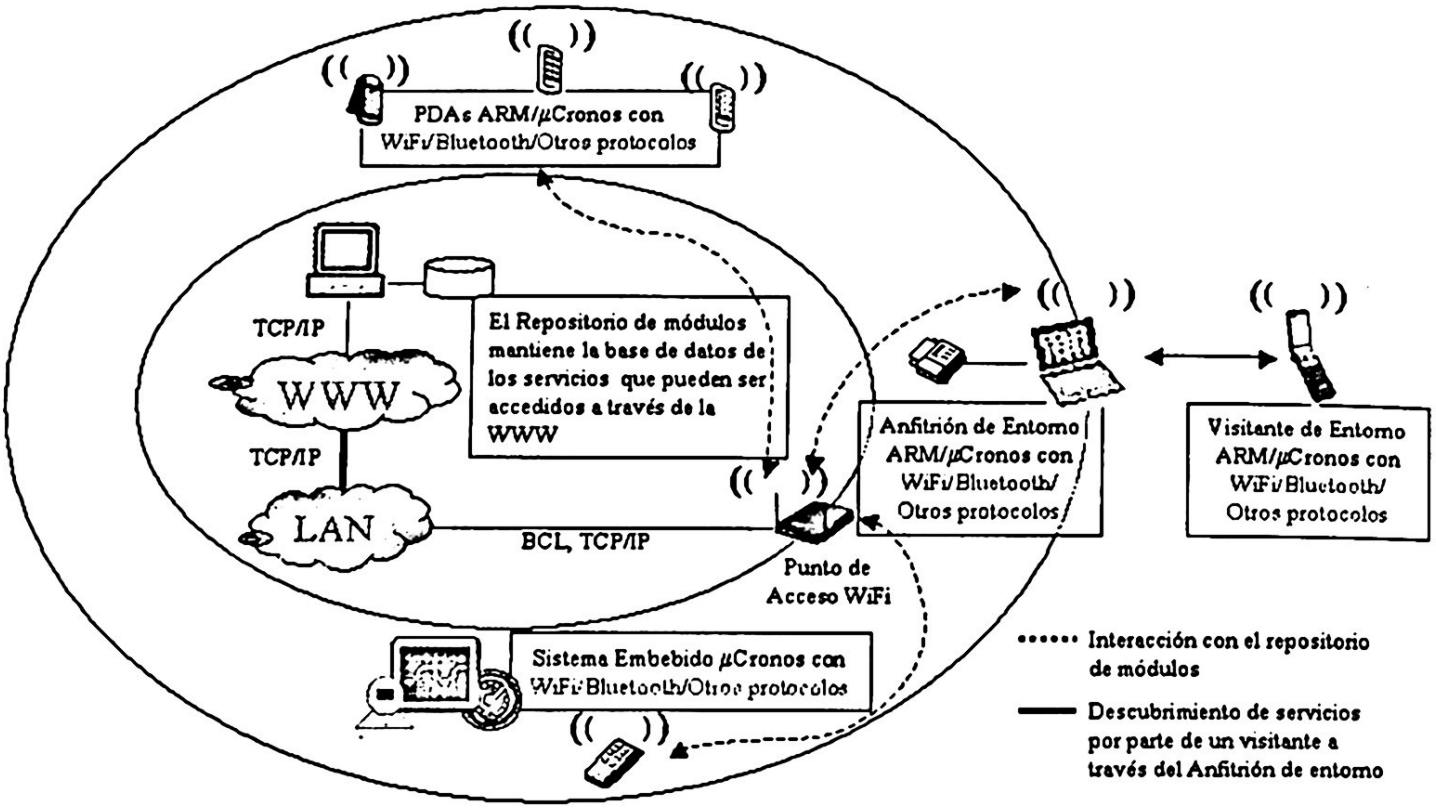

Figura 2. Entorno Ubicuo Oficina.

También en la misma Fig. 2 podemos observar un escenario típico de la utilización de μCronos, donde dispositivo móvil denominado como "Visitante del Entorno" carga (o se deshace de) nuevos módulos función de las características del entorno que está visitando. La carga de estos módulos se realiza por medio de la Capa de Comunicaciones Básica (BCL) y accediendo un repositorio que almacena dichos módulos.

Estos cambios se deben realizar sin la necesidad de instalación explícita de nuevo software o la reconfiguración explícita de los parámetros de configuración.

Una vez descritas las características de los entornos los que se despliega un sistema de la naturaleza μCronos, en las secciones posteriores describiremos sus elementos arquitectónicos.

Las abstracciones de bajo nivel (mecanismos) son aquellas implementaciones dependientes de la arquitectura del sistema, mientras que las abstracciones de alto nivel (políticas) pueden ser implementadas de forma independiente y ser portables a diversas arquitecturas.

Un sistema operativo extensible difiere de un sistema operativo tradicional en que éste es únicamente un esqueleto, el cual puede aumentarse con módulos específicos para extender su funcionalidad.

Un sistema operativo extensible es capaz de agregar dinámicamente nuevos servicios o adaptar los ya existentes basado en las cambiantes demandas de las aplicaciones sin comprometer la estabilidad del sistema [12].

Dada esta definición podemos identificar claramente los tres elementos que comprenden un sistema operativo extensible:

1. Un micro núcleo que proporciona un conjunto de servicios básicos.
2. Mecanismos de extensión que permiten agregar nuevos servicios en función de los servicios primitivos básicos.
3. Una colección de módulos que pueden agregarse al micro núcleo, usando mecanismos de enlace dinámico.

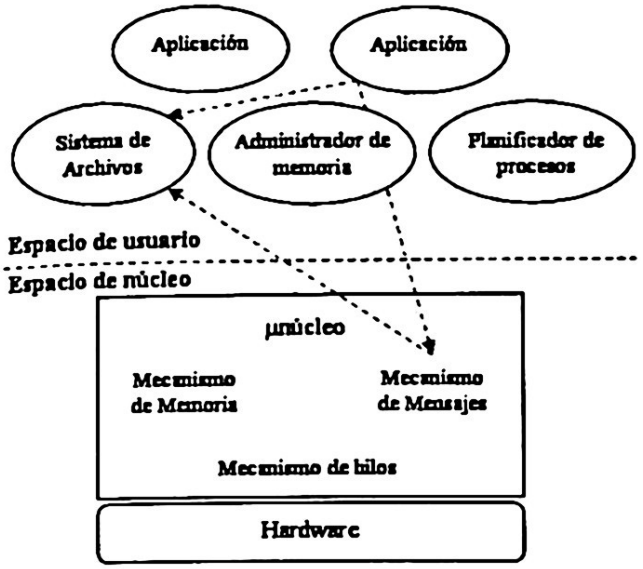

Figura 3. Arquitectura micro núcleo.

4. CAPA BÁSICA DE COMUNICACIONES

En µCronos los sockets pueden ser implementados sobre una Capa de Comunicaciones Básica (BCL), evitando pasar por la pila de protocolos TCP/IP. De esta manera se elimina la sobrecarga en las comunicaciones, pudiendo entonces aprovecharse prácticamente el ancho de banda total para las aplicaciones especificas.

Sin embargo, como se muestra en la Fig. 5, utilizar una BCL no significa romper con los protocolos TCP/IP, puesto que estos son necesarios para interaccionar con sistemas heterogéneos, pero si dar una alternativa para que los sistemas puedan aprovechar las ventajas que una BCL ofrece como la posibilidad de hacer eso eficiente del ancho de banda restringido y propenso a errores propio de los sistemas móviles. La implementación de esta BCL se realiza basándose en el soporte básico de comunicaciones para µCronos [13], donde se describen detalles de implementación para Tarjetas de Interfaz de Red (NIC).

La NIC debe ser inicializada antes de poder iniciar las comunicaciones. A continuación se muestra, en la Fig. 4, la implementación de la inicialización que es valida para chips compatibles con el estándar NE2000 que hemos usado en nuestras pruebas.

```
// seleccionar pagina 0, detener NIC
outp(PuertoES, 0x21);
// FIFO, comando enviar ejecutado, operación normal, DMA 16
bits, big // endian, word configuración de datos
outp(PuertoES + 0xe, 0x59);
// Establecer registros del contador de bytes de datos remotos = 0
outp(PuertoES + 0xa, 0x0);
outp(PuertoES + 0xb, 0x0);
// Establecer registro de configuración de la recepción (RCR)
outp(PuertoES + 0xc, 0x1c);
// TPSR - Registro Inicio de Pagina para Transmisión
outp(PuertoES + 0x4, 0x7a);
// TCR - Registro de Configuración de la Transmisión
outp(PuertoES + 0xd, 0x2);
// PSTART - Registro Inicio de Pagina.
outp(PuertoES+1,0x40);
// BNRY - Registro de Limite
outp(PuertoES+3,0x40);
// PSTOP - Registro Fin de Pagina
outp(PuertoES+2,0x79);
// Registro de Comandos 0x0
outp(PuertoES,0x61);
// CURR - Registro de Pagina Actual
outp(PuertoES+7,0x41);
// Registro de Comandos 0x0
outp(PuertoES,0x21);
// ISR - Registro de Estado de Interrupciones
outp(PuertoES+7,0xff);
// IMR - Registro de Mascara de Interrupciones
outp(PuertoES+0xf,imr);
// TCR - Registro de Configuración de la Trasmisión 0xd
outp(PuertoES+0xd,0);
// RC - Registro de Comandos 0x0
outp(PuertoES,0x61);
// MAR0-7 - Registros de Dirección Multicast
outp(PuertoES+8,0xff);
outp(PuertoES+0x9,0xff);
outp(PuertoES+0xa,0xff);
outp(PuertoES+0xb,0xff);
outp(PuertoES+0xc,0xff);
outp(PuertoES+0xd,0xff);
outp(PuertoES+0xe,0xff);
outp(PuertoES+0xf,0xff);
// RC - Registro de Comandos 0x0
outp(PuertoES,0x21);
// activar la IRQ asociada a la tarjeta de red
outp(picport, picval);
// seleccionar página de registro 0, activar NIC
outp(PuertoES,0x22);
```

Figura 4. Arquitectura de µCronos.

Finalmente, la BCL sirve a µCronos como mecanismo para obtener vía red, los módulos que tenga que cargar dinámicamente. En particular, se podría dar el caso de que el módulo cargado sea el responsable de implementar la pila de protocolos TCP/IP.

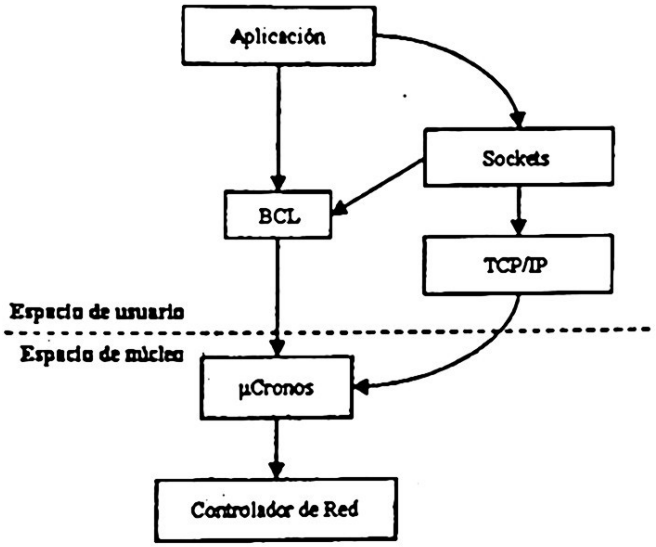

Figura 5. Estructura de comunicaciones BCL vs. Sockets.

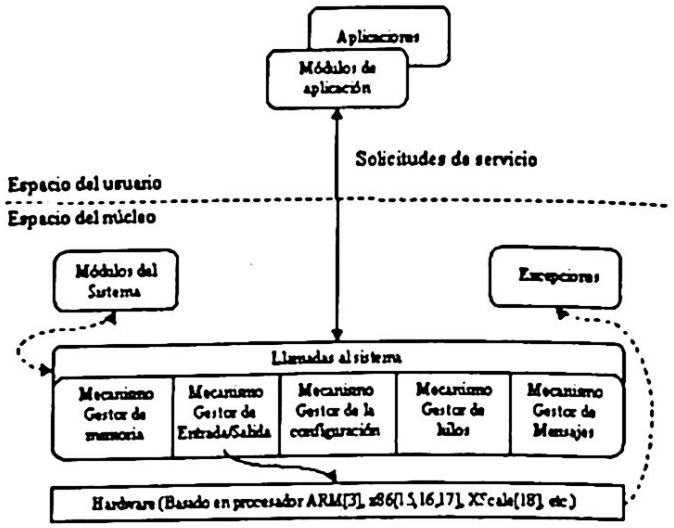

Figura 6. Arquitectura de µCronos.

5. ARQUITECTURA BÁSICA DE µCronos

En la presente sección se realiza una descripción de la arquitectura básica de µCronos y sus principales componentes, ver Fig. 6.

5.1 MECANISMO GESTOR DE MEMORIA

La administración de memoria en µCronos es simple, no se emplea paginación ni intercambio. Esto debido a que está pensado para usarse en sistemas móviles, los cuales generalmente no cuentan con una Unidad de Administración de Memoria (MMU) [14] que dé este soporte. Sin embargo pronto comenzarán a tenerlo.

Los mecanismos para la gestión de la memoria se encuentran estrechamente relacionados con el soporte ofrecido por el procesador y la arquitectura del sistema. Entre las cosas que se tienen que tomar en cuenta para implementar mecanismos gestores de memoria se encuentran:

- Esquema de direccionamiento del procesador
- Modelo de memoria a utilizar
- Soporte de segmentación
- Soporte de paginación

5.1.1 Modelo de Memoria

Este concepto especifica la forma en que el procesador y el sistema operativo administran el espacio de direccionamiento lógico y físico. Existen varios esquemas [19] entre ellos memoria plana, memoria plana páginada, memoria virtual páginada, memoria segmentada, entre otras.

Dado que los sistemas móviles actuales generalmente no cuentan con una MMU, el modelo de memoria que µCronos adopto es el plano. Esto significa que el sistema operativo y las aplicaciones comparten un espacio de memoria lineal en donde las direcciones lógicas son equivalentes a las direcciones físicas. Cada quien tiene una zona de memoria asignada que puede usar para sus propósitos y compartir memoria es muy simple ya que como se puede apreciar en la Fig. 7, no hay restricciones en el direccionamiento, es decir, las aplicaciones y el sistema operativo pueden acceder a donde deseen.

A menos que se utilicé alguna forma de protección, esto puede ser peligroso ya que siempre puede haber un puntero inválido que genere serios problemas. Actualmente la arquitectura de nuestro sistema no contempla ningún mecanismo de protección.

Las principales tareas del gestor de memoria son:

- Inicialización de las estructuras de datos
- Seguir la pista a bloques libres y asignados
- Reducir la fragmentación
- Evitar la relocalización

Existen dos métodos básicos para contabilizar la asignación de memoria física, listas enlazadas y tablas. Para nuestra arquitectura se decidió utilizar listas enlazadas ya que consideramos que es una estructura más dinámica que las tablas, la cual permite manejar bloques de tamaño variable. Esta característica es deseable dada la diversidad de arquitecturas hardware a las que se desea que µCronos sea capaz de adaptarse con el menor esfuerzo posible.

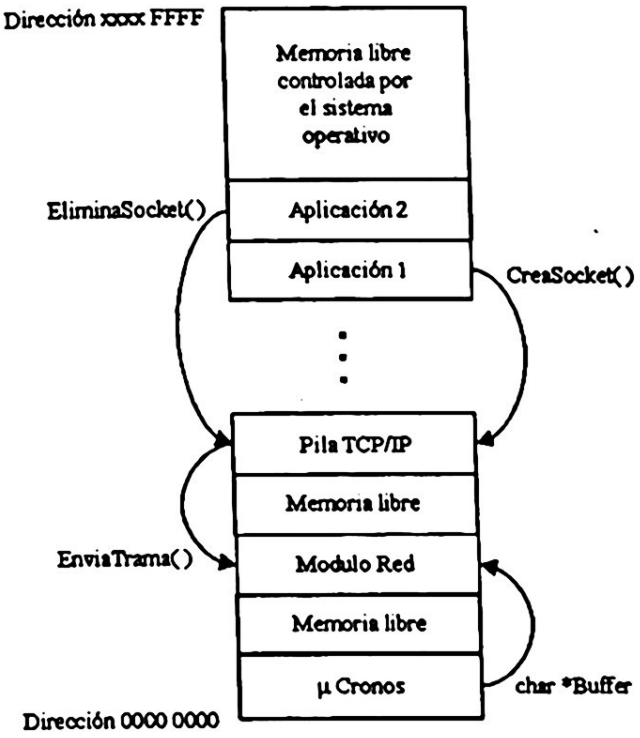

Figura 7. Modelo de memoria plano.

5.1.2 Administración por Listas Enlazadas

La lista enlazada es inicializada por la función ***InicializarListaBloquesLibres***() cuando el sistema obtiene la cantidad de memoria física instalada en el sistema y cada nodo de ésta tiene la estructura que se muestra en la Fig. 8.

```
struct DescriptorBloque
{  unsigned int Base;
   unsigned int Tam;
   unsigned int Libre;
   struct DescriptorBloque *Siguiente;
};
```

Figura 8. Estructura de datos para describir bloques de memoria.

Donde ***Base*** es la dirección donde inicia el bloque de memoria, ***Tam*** su tamaño, ***Libre*** es una bandera que indica si el bloque está libre o no y ***Siguiente*** un apuntador al siguiente nodo; en caso de que sea el final de la lista ***Siguiente*** tendrá el valor de NULO.

5.1.2.1 Inicialización

El procedimiento de inicialización que se muestra en la Fig. 9, consiste en definir la cantidad de memoria física instalada e inicializar el primer descriptor de bloque de memoria para que la describa, iniciando a partir del 1er. Mbyte. Posteriormente se inicializan los demás descriptores con ***Base***=0, ***Tam***=0, ***Libre***=TRUE y ****Siguiente***=NULO. De esta forma nuestra lista queda con un descriptor de bloque que describe la memoria física disponible y muchos descriptores nulos que posteriormente serán utilizados.

```
void InicializarListaBloquesLibres()
{  // apuntador al primer bloque libre
   BloqueInicial = &ListaDeBloquesLibres[0];
   BloqueActual = &ListaDeBloquesLibres[0];
   BloqueFinal = &ListaDeBloquesLibres[NUMEROBLOQUES-1];
   BloqueFinal->Siguiente = NULO;
   // Direcció de inicio del bloque
   BloqueActual->Base = 1024*1024;
   // Tamaño del bloque 256 MB
   BloqueActual->Tam = 256*1024*1024;
   // Bloque libre
   BloqueActual->Libre = TRUE;
   // Apuntador al siguiente bloque libre NULO
   BloqueActual->Siguiente = NULO;
   // Inicializar todos los descriptores de bloques de memoria

   while(BloqueActual < BloqueFinal)
   {  // siguiente descriptor a inicializar
      BloqueActual = BloqueActual+1;
      // Dirección de inicio del bloque 0
      BloqueActual->Base = 0;
      // Tamaño de bloque 0
      BloqueActual->Tam = 0;
      // Bloque libre
      BloqueActual->Libre = TRUE;
      // Apuntador a siguiente bloque NULO
      BloqueActual->Siguiente = NULO;
   }
}
```

Figura 9. Implementación del procedimiento de inicialización.

Una vez que la lista enlazada ha sido inicializada, sistema queda a la espera de solicitudes para asignar y liberar memoria.

5.1.2.2 Asignación de Memoria

La asignación de memoria consiste de la siguiente serie de pasos:

- Buscar en la lista un descriptor libre de tamaño adecuado
- Obtener un descriptor libre
- Guardar en el información del bloque asignado
- Devolver la dirección del bloque asignado

La implementación de estos pasos se muestra en la Fig. 10.

```
// Obtiene una Cantidad especificada de memoria y devuelve
// la dirección donde esta se encuentra

unsigned int ObtenerMemoria(unsigned int Cantidad)
{   struct DescriptorBloque *DescSig;
    struct DescriptorBloque *DescAct;
    DescAct = BloqueInicial;

    while (DescAct != NULO)
    {  // si el bloque esta libre y es de tamaño adecuado

       if ((DescAct->Libre) )

       {  if(DescAct->Tam > Cantidad)

          {  // se obtiene un nuevo descriptor y
             DescSig = NuevoDescriptor();
             // en él se guarda la información del bloque libre
             // menos la cantidad asignada al nuevo descriptor
             // del bloque asignado

             DescSig->Base=DescAct->Base+Cantidad ;
             DescSig->Tam =DescAct->Tam-Cantidad;
             DescSig->Libre = TRUE;
             DescSig->Siguiente = NULO;

             // en el descriptor del bloque libre, se guarda la
             // información del nuevo bloque asignado
             DescAct->Tam = Cantidad;
             DescAct->Libre = FALSE;
             DescAct->Siguiente = DescSig;
          }else

          {  if(DescAct->Tam == Cantidad)
             {  // el descriptor del bloque libre, guarda la
                // información del nuevo bloque asignado
                DescAct->Libre = FALSE;
             }
          }
          return DescAct->Base;
       }
       DescAct = DescAct->Siguiente;
    }
    kprintf("No hay memoria disponible...\n");
    return 0;
}
```

Figura 10. Implementación del procedimiento para asignar memoria.

5.1.2.3 Liberar y Defragmentar Memoria

Para liberar bloques de memoria previamente asignados, se necesita realizar los siguientes pasos:

Buscar en la lista el descriptor del bloque a liberar
Marcarlo como libre
Verificar si existen bloques contiguos libres
Si existieran, fusionar sus descriptores

La correspondiente implementación de estos pasos se muestra en la Fig. 11. Es importante mencionar que el proceso de defragmentación de la memoria se puede realizar concurrentemente con cualquier otro proceso de usuario o del sistema operativo.

```
// Libera el descriptor de un bloque de memoria previamente, la función
// verifica si el descriptor a su derecha describe a un bloque de
// memoria libre, en cuyo caso procede a fusionar ambos descriptores

unsigned int LiberarMemoria(unsigned int Direccion)
{   struct DescriptorBloque *DescAnt,
    struct DescriptorBloque *DescAct;
    struct DescriptorBloque *DescSig;

    DescAnt = NULO;
    DescAct = BloqueInicial;

    while (DescAct != NULO)
    {  // Si se encuentra descriptor ocupado, con Base = Direccion
       if ((!DescAct->Libre) && (DescAct->Base == Direccion))
       {
          // indicar que ahora es un descriptor de memoria libre
          DescAct->Libre = TRUE;

          // verificar si el siguiente descriptor de memoria esta
          // marcado como libre para fusionarlo
          DescSig = DescAct->Siguiente;

          // Si el descriptor anterior esta libre fusionar con el actual
          if ((DescAnt != NULO)&&(DescAnt->Libre))
          {  // fusionando descriptor actual con descriptor anterior
             DescAnt->Tam=DescAct->Tam+DescAnt->Tam;
             DescAnt->Libre = TRUE;
             DescAnt->Siguiente = DescAct->Siguiente;

             DescAct->Base = 0;
             DescAct->Tam = 0;
             DescAct->Libre = TRUE;
             DescAct->Siguiente = NULO;

             // El descriptor anterior es ahora el actual, al
             // fusionarse ambos descriptores
             DescAct=DescAnt;
          }

          // Si descriptor siguiente esta libre, fusionar con el actual
          if ((DescSig->Libre) && (DescSig!=NULO))
          {  // fusionando descriptor actual con descriptor siguiente
             DescAct->Tam =DescAct->Tam+DescSig->Tam;
             DescAct->Siguiente = DescSig->Siguiente;
             DescSig->Base = 0;
             DescSig->Tam = 0;
             DescSig->Libre = TRUE;
             DescSig->Siguiente = NULO;
             kprintf("Fusionando bloques libres ...\n");
          }
          return 1;
       }
       DescAnt = DescAct;
       DescAct = DescAct->Siguiente;
    }
    return 0;
}
```

Figura 11. Implementación del procedimiento para liberar memoria.

Cabe remarcar que este código incluye lógica para evitar la fragmentación de memoria, además se piensa mejorar el algoritmo usando dos listas enlazadas, una que maneje bloques de tamaño menor o igual a 2 Kb y la otra que maneje bloques

mayores de 2 Kb. Esto se piensa puede reducir en buena medida la fragmentación externa.

5.2 MECANISMO GESTOR DE E/S

Debe presentar una interfaz homogénea [20] que pueda ser extendida y modularizada a fin de soportar una amplia variedad de dispositivos, con servicios básicos como los mostrados en la tabla 1.

TABLA 1. SERVICIOS ESTÁNDAR DE ENTRADA/SALIDA

Servicio	Función
Abrir	Establecer un descriptor que asocie dispositivos a procesos con fines de administración.
Cerrar	Liberar dispositivo de cierto proceso al que previamente fue sido asignado.
Leer	Mecanismo básico para leer de algún dispositivo de E/S.
Escribir	Mecanismo básico para escribir de algún dispositivo de E/S.
ControlES	Configuración del modo de operación a bajo nivel del dispositivo físico.

5.2.1 Controladores de Dispositivo

La habilidad del sistema para soportar dispositivos futuros está basada en la definición de una interfaz.

Estos servicios son llamados puntos de entrada, se encuentran implementados en el controlador de dispositivo y definen el conjunto de operaciones que el sistema operativo puede realizar con los dispositivos.

5.3 MECANISMO GESTOR DE LA CONFIGURACIÓN

Es responsable de cargar, descargar y enlazar módulos sobre demanda; actuando como un repositorio de interfaces [21]. Cuando los módulos son cargados, ellos registran sus interfaces y sus contratos de ambiente.

5.3.1 Módulos

Son componentes de software que proveen servicios, los cuales se pueden exportar a otros módulos con la idea de colaborar. Pueden realizar servicios en el nivel de aplicación o en el de sistema.

Como puede apreciarse en la Fig. 6, pueden residir en el espacio del usuario o del sistema, asumiendo que estos conceptos (espacio de usuario y de sistema) ya fueron previamente definidos por otros módulos. No tienen servicios de intercomunicación por defecto, se comunican quienes tienen enlaces.

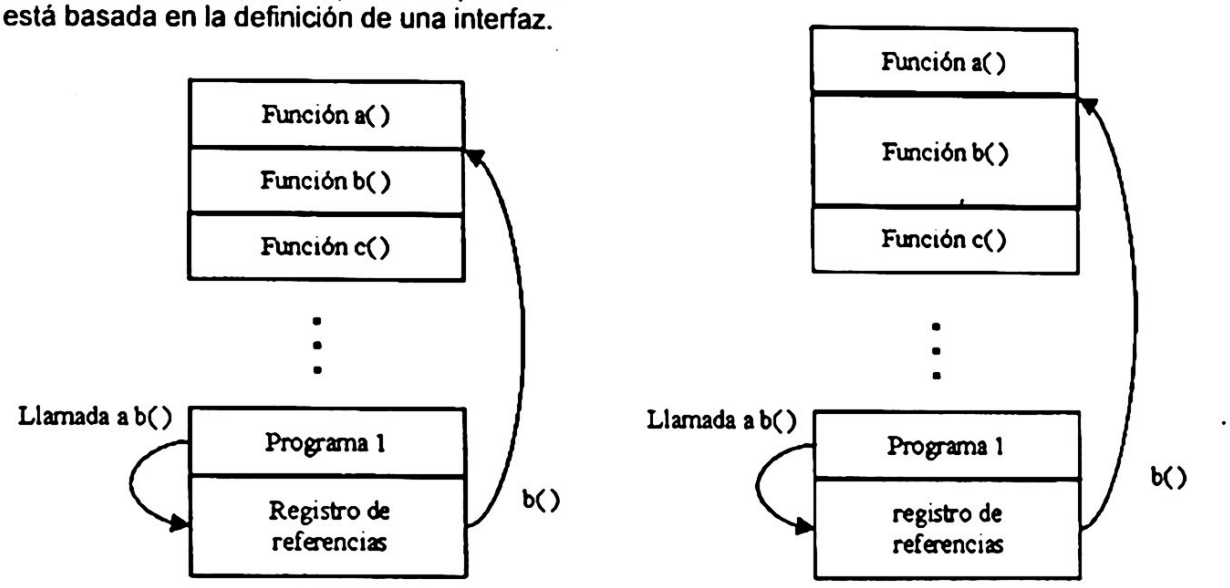

Figura 12. Actualización del registro de referencias.

El sistema no tiene idea de las operaciones que puede realizar determinado dispositivo pero es capaz de averiguarlo por medio de la interfaz y los servicios estándar como Abrir, Cerrar, Leer, Escribir, ControlES, etc.

Tampoco existen modelos de memoria predefinidos como por ejemplo memoria plana o paginada, se espera que estos sean definidos por otros módulos que forman parte de un ambiente de ejecución específico.

5.3.2 Enlace de Módulos

responsabilidad del Mecanismo Gestor de la Configuración validar una solicitud de enlace entre dos módulos, esto se hace verificando las interfaces, tipos de datos y evaluando sus contratos ambiente.

sistema controla la creación de enlaces y verifica que los módulos se enlacen correctamente, es decir, un módulo únicamente puede invocar los servicios de otro módulo si se cumplen las condiciones definidas en los contratos de ambiente. Esto incluye tanto a los módulos de software puro, como a aquellos que encapsulan hardware.

Una vez enlazados, los módulos se invocan de forma correcta sin importar las futuras reubicaciones de los mismos [22,23]. Esto puede verse en la Fig. 12.

5.3.3 Contratos de Ambiente

Permite agregar restricciones de seguridad a módulos con el fin de controlar quiénes y bajo qué condiciones pueden ser enlazados. Las restricciones de seguridad son predicados de forma nombre–valor que deben ser verdaderos para realizar el enlace. Este proceso es realizado en tiempo de enlace y se considera que no afecta significativamente el rendimiento total del sistema.

5.4 MECANISMO GESTOR DE HILOS

continuación se describirán las características principales de los hilos propuestos en nuestra arquitectura y los estados por los que pasan durante su vida en el sistema. Un proceso es considerado un contenedor de recursos, en el cual se encuentra el espacio de direcciones virtual e información de control necesaria para la ejecución de uno o más hilos de ejecución.

Un hilo consiste esencialmente del estado actual del procesador y un espacio de pila. La ventaja de usar hilos es que comparten los recursos de su proceso padre, varios hilos pueden sincronizarse y colaborar entre si, y su cambio de contexto es mucho más rápido que el de un proceso. Un cambio de contexto obtiene el contenido actual de los registros del procesador y los guarda en una instancia de hilo para su posterior uso.

5.4.1 Estados de los Hilos

Los hilos una vez creados, forzosamente tienen que colocarse en algún lugar mientras esperan a ser procesados. Dependiendo de sus acciones es necesario formarlos en diferentes colas, cada cola representa un estado distinto. Como se ilustra en la Fig. 13, básicamente existen cinco estados por los que puede pasar un hilo en μCronos.

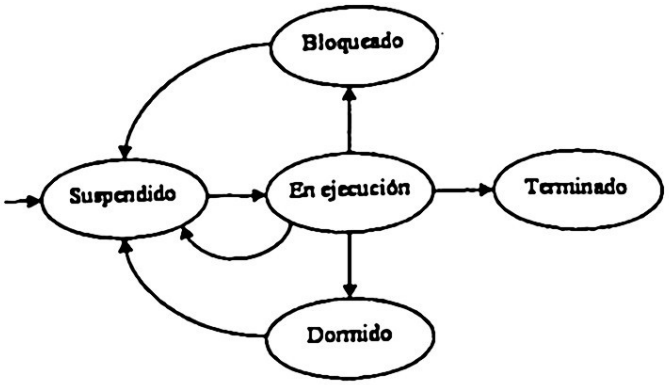

Figura 13. Estados de hilos.

5.4.2 Características de los μHilos

Nuestros hilos cuentan con una estructura muy sencilla, que permite realizar cambios de contexto eficientes y que no imponen una gran sobrecarga de procesamiento. También se provee un conjunto completo de funciones para la gestión de los hilos de forma eficiente.

Tabla 2. Componentes y servicios con que los μhilos deben de contar.

Componentes	Servicios
ID	Crear
Contexto	Establecer contexto
Pila	Obtener contexto
Estado	Establecer información
Prioridad base	Obtener información
Prioridad dinámica	Iniciar
Tiempo de ejecución	Suspender
Contador de suspensión	Reanudar
	Terminar

5.4.2.1. Estructura del Bloque de Control de Hilo

Contiene información necesaria para el correcto funcionamiento de los hilos. Información como los registros generales del procesador, banderas, apuntadores de instrucción, pila y segmentos adicionales que forman el contexto o estado del procesador.

Una zona de memoria correspondiente a su pila es indispensable para la operación de un hilo, otra información como número de identificador y estado, forman la información administrativa importante para gestionar los hilos en el sistema, ver Fig. 14.

```
typedef struct
{  unsigned int edi;
   unsigned int esi;
   unsigned int ebp;
   unsigned int esp;
   unsigned int ebx;
   unsigned int edx;
   unsigned int ecx;
   unsigned int eax;

   unsigned int ds;
   unsigned int es;
   unsigned int fs;
   unsigned int gs;

   unsigned int NumeroINT;
   unsigned int CodigoDeError;
   unsigned int eip;
   unsigned int cs;
   unsigned int eflags;
} registrosx86;


typedef struct
{  // identificador de hilo
   int id;
   // estado de los registros del procesador
   registrosx86 contexto;
   // Pila de 1024 enteros para cada hilo
   unsigned int Pila[1024];
   // 0 = inicializado, 1 = listo , 2 = ejecución
   int estado;
} BCH;
```

Figura 14. Estructura Bloque de Control de Hilos.

5.4.2.2. Cambios de Contexto

Esta operación es muy importante para el desempeño general del sistema ya que se ejecuta repetidamente durante la ejecución del mismo. La conmutación de contexto, forma parte del conjunto de mecanismos del SO debido a que tiene que acceder directamente a elementos de bajo nivel como registros del procesador, banderas, etc.

Esta función, ver Fig. 15, es parcialmente implementada en ensamblador y por tanto parte del código dependiente de la arquitectura. Se vale de la estructura BCH (Bloque de Control de Hilos) mostrada en la Fig. 14 para obtener el contexto y datos administrativos necesarios para la conmutación.

5.5 MECANISMO GESTOR DE MENSAJES

El mecanismo de comunicación entre hilos es importante dentro del sistema operativo porque notifica la ocurrencia de eventos, lo cual ayuda a la sincronización de las actividades y permite compartir información entre hilos aún a través de las barreras de protección de memoria que pudieran existir.

```
if (BloqCtrlHilo[SigHilo].estado == 0)
{  // estado en ejecución
   BloqCtrlHilo[SigHilo].estado = 1;

   // guardar esp del hilo detenido
   ESPHiloDetenido = ESP;

   // obtener valor actual del registro ESP
   s = (unsigned int *)ESP;

   // eliminar variables locales
   s=s+12;


   DirRetornoFuncion=(int)s;
   BloqCtrlHilo[ActHilo].contexto.esp= DirRetornoFuncion;

   // con los anteriores incrementos *s apunta a la dirección
   // de retorno de la función actual

   ESPSiguienteHilo = BloqCtrlHilo[SigHilo].contexto.esp;

   // conmutar a la pila del siguiente hilo
   ESP = ESPSiguienteHilo;

   // visualizar datos en la pila del siguiente hilo
   s = (unsigned int *)ESP;

   pulso++;

   // resetear chip de interrupciones 8259
   outportb(0x20, 0x20);

   __asm__ __volatile__(
      // recuperar contexto nuevo de la pila y
      // brincar hacia el

      "popa;"
      "pop %%ds;"
      "pop %%es;"
      "pop %%fs;"
      "pop %%gs;"
      "add $8, %%esp;"
      "iret;"
      : :
   );
}
```

Figura 15. Implementación del cambio de contexto.

En nuestro caso utilizaremos una forma muy eficiente y adaptable de mensajes basada en las funciones estándar Enviar() y Esperar(). Como se muestra en la Fig. 16, son implementadas usando semáforos y memoria compartida [19]. Esto se debe a que en nuestro modelo de memoria, ver Fig. 7, no existe protección[1]. Por otro lado, la memoria compartida es la opción más eficiente y sencilla de implementar.

[1] Recordemos que la mayoría de las arquitecturas móviles y embebidas no dan soporte para ello.

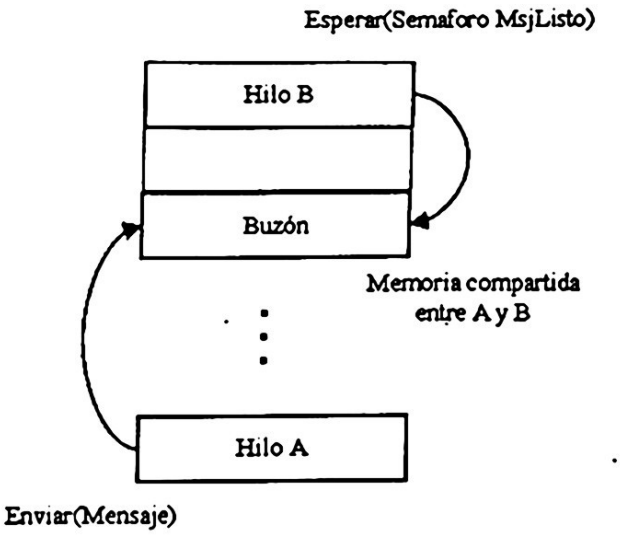

Figura 16. Sistema de mensajes.

6. CONCLUSIONES Y TRABAJOS FUTUROS

Es común, que al desarrollar sistemas de propósito general, se provea de funcionalidad que en realidad no es necesaria, sin embargo, esta funcionalidad extra innecesaria consume recursos que si son necesarios para la actividad particular deseada. Esto es un problema de suma importancia en las aplicaciones móviles y en los sistemas embebidos donde los recursos son limitados. Obviamente las abstracciones de alto nivel tienen sus ventajas, pero muchas veces se subutiliza la capacidad real de nuestros equipos debido a la funcionalidad extra. Por lo tanto, proponemos una arquitectura que permita a los sistemas móviles y embebidos modificar transparente y dinámicamente su sistema operativo respondiendo a cambios de su entorno. De esta manera, el sistema únicamente tendrá la funcionalidad que es estrictamente necesaria.

Se espera que esta arquitectura sirva de plataforma para el desarrollo de sistemas ubicuos como los descritos por Weiser.

Agradecimientos: Se agradece a los revisores por sus comentarios. Al CIC-IPN y al CONACyT, por el apoyo al desarrollo de esta investigación.

7. REFERENCIAS

[1] Alistair C. Veitch. "A Dynamically Extensible and Configurable Operating System", http://www.hpl.hp.com/personal/Alistair_Veitch/papers/thesis/thesis.pdf, Ph.D.thesis, Computer Science,University of British Columbia,June,1998

[2] Eric S. Raymond. , "The Cathedral and the Bazaar", (http://www.tuxedo.org/~esr/writings/cathedral-bazaar/)

[3] Home page ARM Ltd., www.arm.com

[4] Steve Furber, "ARM System-on Chip Architecture", Addison Wesley Longman 2000

[5] Weiser, M., "The Computer for the 21st Century", Scientific American, Vol. 265, No. 3, September 1991.

[6] Satyanarayanan, M., "A Catalyst for Mobile and Ubiquitous Computing", IEEE Pervasive Computing, Vol. 1,No. 1, January-March 2002.

[7] Kindberg, T. and Fox, A., "System Software for Ubiquitous Computing", IEEE Pervasive Computing, Vol. 1, No. 1, January-March 2002.

[8] IEEE 802.11b Working Group for Wireless LANS, http://grouper.ieee.org/groups/802/11/index.html

[9] Bluetooth Consortium web site, http://www.bluetooth.org

[10] Michael Gien., "Key to Modern Operating Systems Design", Chorus systémes, 1990.

[11] Sun Microsystems, "ChorusOS 5.0 features and Architecture Overview", December 2001.

[12] Leendert van Doorn, "The Design and Application of an Extensible Operating System", Labyrint Publication (ISBN 90-72591-88-7)

[13] Victor D. Castillo y Rolando Menchaca, "Implementación del Soporte Básico de Comunicaciones para µCronOS", www.geocities.com/ViDaSoftwareInc/CronOS/BCL.pdf Centro de Investigación en Computación-IPN, México D.F.

[14] Descripción del proyecto µCLinux, http://www.uclinux.org/description/

[15] Intel Corp., (http://www.intel.com/design/pentium4/manuals/245471.htm), "IA32 Intel® Architecture Software Developer's manual. Vol.1 - Instruction Set Reference"

[16] Intel Corp., (http://www.intel.com/design/pentium4/manuals/245472.htm), "IA32 Intel® Architecture Software Developer's manual. Vol.3 - System Programming Guide"

[17] Barry B. Brey, "Los microprocesadores Intel 8086/8088, 80186, 80286, 80386 y 80486 – Arquitectura, Programación e Interfaces" 3ra. Edición, Ed. Prentice Hall Hispanoamericana

[18] Home page Intel XScale, http://www.intel.com/design/intelxscale/

[19] Richard A. Burgess, "Developing Your own 32 Bit Operating System" (ISBN 0-672-30655-7), 1ra. Edición SAMS/MacMillan.

[20] Jason Spencer, "I/O Software Architecture in Modern Operating Systems" , 1rst. edition, Mindshare, Inc.

[21] Michael Clarke y Geoff Coulson "An Architecture for Dynamically Extensible Operating Systems" , 4 th ICCDS, Annapolis, Maryland, May 1998.

[22] Galderic Punti Marisa, Gil Xavier Martorell y Nacho Navarro, "gtrace: function call and memory access traces of dynamically linked programs in IA-32 and IA-64 Linux", reporte numero: UPC-DAC-2002-51, Universidad Politécnica de Cataluña

[23] Marisa Gil, Xavier Martorell y Nacho Navarro, "Function Call Traces in Separate Linked and Loaded Modules" reporte numero: UPC-DAC-2002-19, Universidad Politécnica de Cataluña