Picix: Un microkernel híbrido preemptivo/cooperativo

25 de julio de 2003

Artículo enviado al IX Congreso Argentino de Ciencias de la Computación

ALFREDO ADRIÁN ORTEGA, Licenciado en Informática
Universidad Nacional de la Patagonia San Juan Bosco, Facultad de Ingeniería, Sede Trelew
Dirección: Los Aromos 110, Trelew, Chubut, C.P.9100
Email: aortega@mailandnews.com

Resumen:
Se propone un sistema operativo capaz de funcionar en sistemas de bajos recursos, tales como los microcontroladores modernos. Se describen varios servicios y en especial su scheduler híbrido, que puede ser utilizado tanto en modo preemptivo como cooperativo. Se presenta una implementación en lenguaje C para la línea de microcontroladores PIC-18 de Microchip.

Palabras Clave:
Sistema Operativo, embebido, microkernel, tiempo real, preemptivo, cooperativo

1. Introducción

Durante el desarrollo del trabajo final de grado del autor surgió la necesidad de un sistema operativo capaz de ofrecer multi-programación sobre sistemas de microcontroladores, ofrecer determinados servicios y funcionar con pocos recursos. Dicho sistema fue diseñado e implementado para soportar una aplicación de Firewall, que era el objetivo del proyecto. Éste artículo se enfoca sobre el diseño del kernel, que presenta un comportamiento que podría considerarse interesante, ya que aunque existen a la fecha varios sistemas operativos disponibles para la plataforma de hardware seleccionada, no todos poseen la flexibilidad del que se describe en éste artículo.

> Nota: No se debe confundir la denominación de *'Microkernel'*, utilizada durante el artículo actual para acentuar el caracter de reducida funcionalidad, con el término utilizado en [tanenbaum:1992], usado para referirse a una arquitectura modular de sistemas operativos. El sistema aqui descrito es monolítico.

2. Clasificación de Sistemas Operativos

La multi-programación de un sistema permite que varios procesos o hilos corran de manera concurrente, al contrario de sistemas denominados "batch" en donde todos los procesos son colocados en una cola y ejecutados consecutivamente (ver Tanenbaum:1992). Pero para lograr la sensación de simultaneidad al poseer sólo un CPU, cada proceso debe correr un cierto tiempo y luego entregar el control a otro proceso. Ahora surgen dos comportamientos bien definidos que pueden seguirse:

Cooperativo: Permitir que cada proceso entregue el control al sistema operativo cuando lo considere necesario. Por lo que todos los procesos deben *cooperar* para así repartirse los recursos del sistema. Una desventaja es que un proceso 'egoista' que se apropia de algún recurso como lo es el CPU no puede ser interrumpido, y su mal funcionamiento afecta a todo el sistema. Como ejemplo, existe Windows 3.1, sistema operativo de la empresa Microsoft.

Preemptivo: En éste caso el sistema operativo suspende temporalmente a todo proceso que considere que haya recibido suficiente tiempo de CPU. Para realizar esta tarea se necesita soporte de hardware, que debe proveer de algún medio para quitar el control a un proceso, como puede ser una interrupción de hardware. Windows 95 es un ejemplo típico de un sistema de éste tipo.

En sistemas embebidos se suelen utilizar sistemas del tipo cooperativo, por ofrecer mayor control sobre el hardware. Por otra parte, los sistemas preemptivos facilitan la producción de software mas complejo, ya que el programador no debe preocuparse del cambio de contexto.

El sistema desarrollado, que de ahora en adelante se denominará como *picix*, corresponde a una clase llamada sistemas híbridos, ya que su comportamiento puede ser tanto cooperativo como preemtivo. Esto otorga mayor flexibilidad al programador, que puede elegir el comportamiento deseado, y modificar dicho comportamiento durante el funcionamiento del sistema.

3. Plataforma seleccionada

El sistema microcontrolador (modelo PIC18F452, de la línea PIC-18 de Microchip) utilizado como plataforma de desarrollo posee un hardware un tanto atípico [UYM:2000], que se describe a continuación:

- Arquitectura Hardvard, que difiere de la arquitectura Von-Neumann más extendida.
- Conjunto de instrucciones RISC (Reduced Instruction Set CPU).
- Memoria de datos separada de la memoria de programa.
- Memoria linealmente accesible y carencia de sistemas de protección.
- Pila de llamadas (call...ret) implementada en hardware, modificable programáticamente.
- Varias fuentes de interrupción.

Estas características disponibles permiten realizar un sistema operativo preemptivo, aunque no se puede lograr una separación entre procesos, y ni siquiera se puede aislar efectivamente los procesos del kernel. Esto puede presentar un problema en sistemas muy complejos, ya que un proceso podría acceder directamente las estructuras del kernel y causar un mal funcionamiento, aunque generalmente el programador tiene el control total sobre el software que corre sobre un microcontrolador, por lo que una revisión cuidadosa del sistema podría subsanar estos problemas.

4. Características del sistema

Las siguientes metas fueron fijadas desde un principio, al implementar el sistema operativo picix:

- Desarrollar un sistema preemptivo multi-hilo
- Implementar delays (Retrasos y llamadas bloqueantes)
- Estadísticas de utilización del procesador
- Implementación en el lenguaje C de la mayor parte del sistema.

No se diseño ningún tipo de sistema IPC (Inter Process Comunication, Comunicación Inter-Proceso), ya que la aplicación para la cual fue diseñado el sistema picix no lo requiere, pero podría ser una adición futura.

Otra característica importante del sistema es que no se puede hablar de procesos en el sentido utilizado en sistemas Unix o Windows, ya que como se dijo anteriormente, no es posible el aislamiento requerido, pero puede lograrse una implementación de hilos con un comportamiento similar a los '*threads*' o hilos de sistemas operativos mayores.

También se desarrolló un sistema de archivos y una pila TCP/IP, pero los detalles de dichos sistemas exceden el alcance de éste artículo.

Listado 1: OS-switch, guarda el contexto actual

```
void OS_switch(void)

{

1.Desactiva las interrupciones: Esto es necesario porque sino si se
    interrumpe el microcontrolador en el medio de un cambio de contexto
    puede quedar en un estado inconsistente.

2. Guardar el contexto:
   Guardar registros especiales.
   Guardar punteros a la pila de Software
   Guardar pila de Hardware

3.Recolectar estadísticas: Se recolectan datos útiles para efectuar el
    debugging, tales como tamaño máximo de las pilas, porcentaje de
    utilización del microcontrolador, etc.

4.Llamar a OS_sched para cargar el siguiente proceso.

}
```

5. Funciones del Scheduler

El scheduler es la porción del sistema que selecciona el siguiente proceso a ejecutar. Se compone de varias funciones, que dependen de la funcionalidad ofrecida. En el sistema picix, el scheduler se sirve de las siguientes funciones, todas implementadas en el lenguaje C.

1. `OS_switch()`: Guarda el contexto de un proceso, como se ve en el pseudo-código del listado 1
2. `OS_sched()`: Carga el contexto del nuevo proceso a ejecutar, detallado en el listado 2
3. `selectNextProcess()`: Llamada por `OS_sched()`, selecciona el siguiente proceso a ejecutar, en base a un algoritmo simple de prioridades (En las primeras versiones, se utilizaba el conocido *round-robin*, en el que todos los procesos tenían la misma prioridad).

`OS_sched()` retorna directamente al punto donde se llamó a `OS_switch()` en un cambio de contexto anterior, o al comienzo del proceso, si es que se lo llama por primera vez.

Listado 2: OS-sched, selecciona el próximo proceso a correr y carga el contexto del mismo

```
void OS_sched(void)

{

1.Selecciona el siguiente proceso a correr.

2.Restaura el contexto:
Restaura la pila de hardware
Restaura los registros especiales
Restaura los punteros a la pila de Software

3.Activa las interrupciones

}
```

6. Contexto

Se denomina contexto a todos los datos necesarios para la ejecución de un proceso. Durante el proceso llamado '*Cambio de Contexto*' se almacena el contexto del proceso actual en una estructura de datos, para luego ser restaurada cuando el scheduler seleccione nuevamente al proceso para ser ejecutado. Los datos que forman el contexto de un proceso están muy relacionados con el hardware y el lenguaje utilizado, ya que diferentes arquitecturas poseen diferentes datos. El contexto para el microcontrolador PIC-18 utilizando el lenguaje C se compone de:

Registros: No hace falta guardar todos sino unos 10 de los aproximadamente 90 que posee. Se recuerda que a diferencia de arquitecturas x86, los dispositivos RISC suelen contar con multitud de registros.

Pila de Hardware: Almacena las direcciones de retorno de hasta 32 llamadas anidadas. En una arquitectura x86, las direcciones de retorno se almacenan en la denominada *pila de software*, que se describe a continuación.

Pila de Software: Es una región de memoria donde se guardan los parámetros, las variables locales (llamadas 'automáticas' en C, ver Sebesta:1996) y los valores de retorno de las funciones. Esta es una región que depende del lenguaje y compilador seleccionado, ya que por ejemplo, no existe si trabajamos directamente en código ensamblador.

Estos datos son los que el scheduler necesita guardar y restaurar para simular la simultaneidad de los diferentes hilos. Las variable globales utilizadas en las funciones no pertenecen al contexto, ya que son compartidas por todos los hilos.

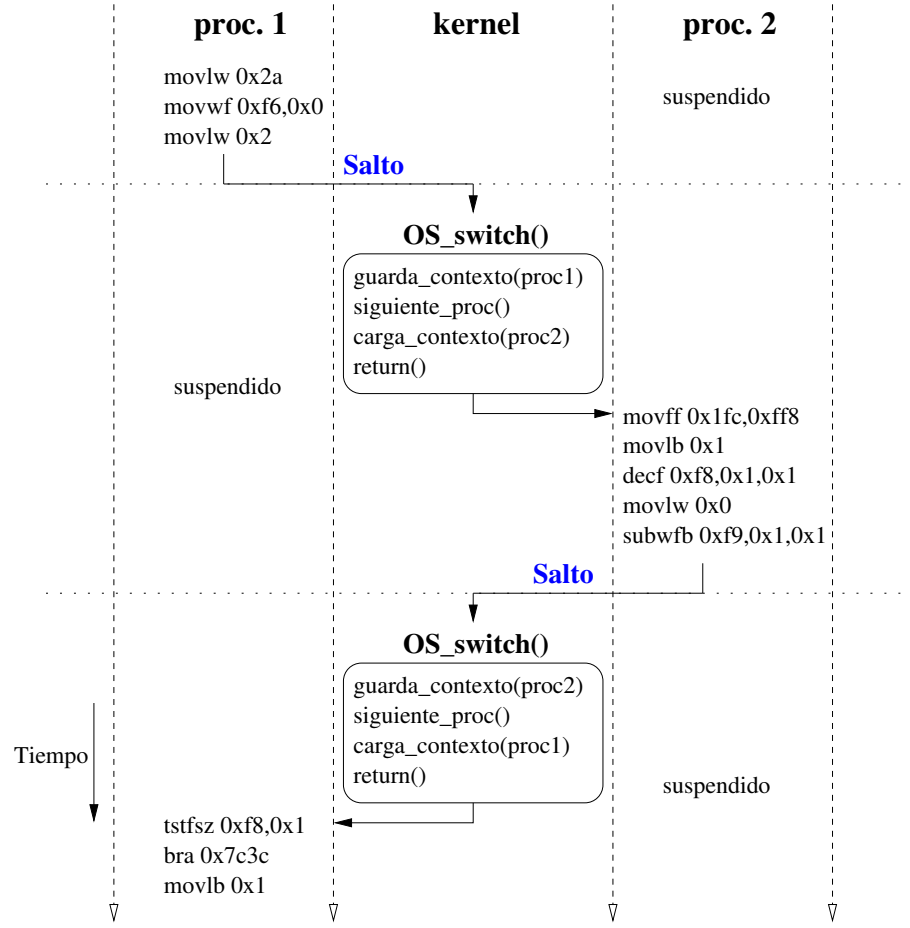

Figura 1: Fucionamiento genérico del cambio de contexto

7. Configuración híbrida

Para lograr un comportamiento preemptivo, se debe llamar a la función `OS_switch()` desde un ISR (Interruption Service Request, pedido de servicio de Interrupción), que realiza el cambio de contexto en cualquier momento de la ejecución de un hilo.

Por el contrario, si es una función de usuario la que llama a `OS_switch()`, se realiza el cambio de contexto exactamente en ese punto, por lo que en éste caso se tiene un sistema operativo cooperativo, cuyo ejemplo se aprecia en el listado 3.

Una combinación de las dos opciones resulta en la mayor flexibilidad, y es llamado sistema operativo preemptivo *mixto* o *híbrido*.

La clave para poder utilizar el sistema en distintas configuraciones es la función `OS_switch()`, que puede realizar un cambio de contexto tanto desde un ISR como desde una función de usuario. En la figura 1, puede observarse el comportamiento del kernel cuando realiza el cambio de contexto entre

Listado 3: Modo cooperativo, ejemplo

```
char cont;

void main(void)
  {
  OS_createProcess(proc1,...);
  while(true)
    {
    cont++;
    OS_switch();
    }
  }

void proc1(void)
  {
  while(true)
    {
    printf(â %i â,cont);
    OS_switch();
    }
  }
```

dos procesos, llamados `proc1` y `proc2`. En la figura se nombra el punto donde se salta a la función `OS_switch()` como "**Salto**". Este salto puede efectuarse de dos maneras:

1. Apuntando un ISR a la función `OS_switch()`, y disparando dicho ISR desde un Timer.
2. Llamando directamente a `OS_switch()` desde cualquiera de los procesos `proc1` o `proc2`. En este caso la función de usuario percibirá a `OS_switch()` como un retardo, cuyo tiempo dependerá de la prioridad del hilo en que se está ejecutando.

El desarrollo se ve facilitado sobre la plataforma PIC-18, que trata de manera similar los saltos a interrupción y las llamadas a función. Quizas deba agregarse código adicional si se desea portar el sistema operativo a otras arquitecturas.

8. Estructuras de almacenamiento

La función `OS_switch()` accede los datos descritos en la sección anterior y los almacena en varias estructuras de datos, cuya definición (en lenguaje C) se aprecia en el listado 4.

Se observa que todos los valores son declarados como '`unsigned char`' debido a que se utilizó un CPU de 8 bits.

La constante `MAX_PROCESOS` indica la mayor cantidad de procesos que el sistema puede mantener.

Listado 4: Definiciones de estructuras de almacenamiento utilizadas

```
/* Registros */
typedef struct {
    unsigned char wreg,bsr,status,prodl,prodh,fsr0h,fsr0l,
                  tblptru,tblptrh,tblptrl,fsr1h,fsr1l,fsr2h,
                  fsr2l,tmp0,tmp1,pclath;
                } registros;

/* Pila de Hardware */
unsigned char Hstkptr[MAX_PROCESOS];
unsigned char HstackL[MAX_PROCESOS][MAX_NIVELES_PILA];
unsigned char HstackH[MAX_PROCESOS][MAX_NIVELES_PILA];

/* Pila de Software */
unsigned char SstackL[MAX_PROCESOS];
unsigned char SstackH[MAX_PROCESOS];
```

La constante `MAX_NIVELES_PILA` indica la profundidad máxima de llamados a función permitidos. Esta constante tiene un límite superior impuesto por el hardware, que no soporta mas de 32 llamadas a funciones anidadas. Esta limitación es común en el ambiente de los microcontroladores, aunque no existe en arquitecturas mayores tales como la x86, que dispone de cantidades de memoria prácticamente ilimitadas.

9. Funciones Adicionales

Se implementaron funciones de apoyo, que son:

1. `OS_init()`: Inicializa variables del sistema y setea el hilo principal de `main()`.
2. `OS_createTask()`: Setea todas las estructuras necesarias para iniciar un proceso.
3. `OS_replace()`: Reemplaza el proceso que lo ejecuta por otro, liberando toda la memoria utilizada por el proceso saliente.
4. `OS_kill()`: Maneja el estado de un proceso, pudiendo pausarlo o reiniciarlo. La eliminación aún no se implementó.
5. `OS_renice()`: Modifica la prioridad de ejecución de un proceso.
6. `OS_rpt()`: Imprime la lista de procesos, junto con la pila de llamadas e información estadística acerca de cada uno.

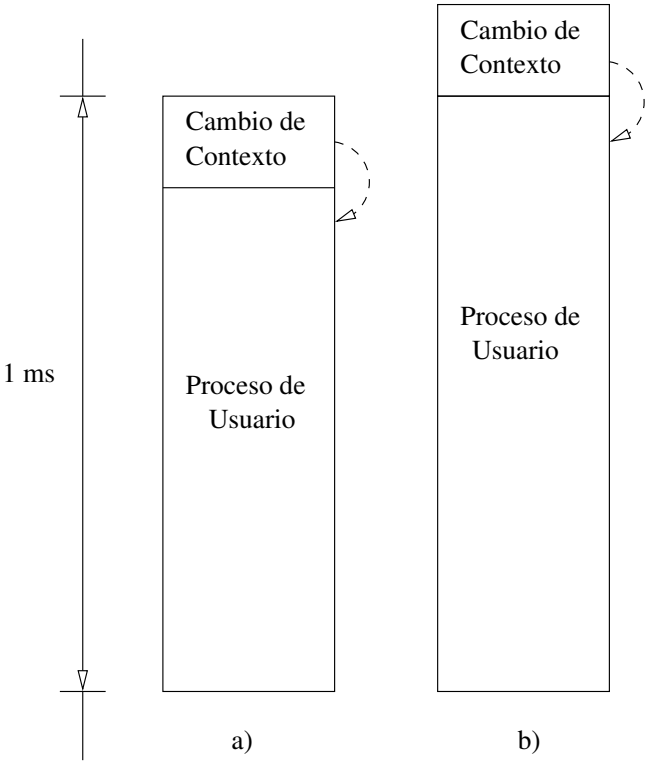

Figura 2: Diferentes configuraciones de Time-Slice

Nota sobre la nomenclatura: El hecho de que las funciones del sistema operativo comiencen con el prefijo "`OS_`" no es fortuito, ya que la mayoría de los sistemas operativos para microcontroladores (Salvo, UC-OS, etc.) llaman a sus funciones de esta manera.

10. Selección de Time - Slice

Cuando el sistema picix se configura en modo preemptivo, puede considerarse que se está dividiendo el tiempo del procesador en "rodajas" (Time-Slices) que luego se asignan a los diferentes hilos de acuerdo a sus prioridades. La granularidad de dicha división debe ser calculada porque si cada "rodaja" es demasiado grande, se perderá la sensación de simultaneidad de los procesos, mientras que si es demasiado chica, se perderá mucho tiempo realizando el proceso de cambio de contexto. El sistema actualmente tiene configurado un Time-Slice de 1 milisegundo (10000 Instrucciones)[1], por lo que cada segundo es dividido en 1000 segmentos que son asignados a los procesos subsecuentemente. Sin embargo, como se ve en la figura 2, existen dos maneras de configurar el scheduler :

a) Asignar 1 milisegundo a la suma del cambio de contexto y el proceso de usuario, o

[1] Valor que en versiones futuras será aumentado, por considerarse demasiado pequeño.

b) Asignar 1 milisegundo exactamente al proceso de usuario, mientras que el cambio de contexto queda fuera del cálculo.

Considerando que el tiempo de cambio de contexto es variable, si se utiliza la opción a) no se garantiza un tiempo constante asignado a cada proceso, mientras que si se utiliza la opción b) sí lo garantiza, pero no se pueden establecer retardos de precisión, debido a que el tiempo de cambio de contexto no es tenido en cuenta para realizar los cálculos.

La elección realizada corresponde a la opción a), para de éste modo poder calcular retardos con precisión.

11. Implementación final

El sistema final consta de sólo 700 líneas de código (El scheduler solamente), y se implementó utilizando el lenguaje C y porciones de ensamblador. El desarrollo fue realizado mediante herramientas provistas por Microchip que se pueden descargar de forma gratuita desde su sitio de Internet http://www.microchip.com.

Se realiza un cambio de contexto en aproximadamente 2800 ciclos de reloj[2], y el código ocupa aproximadamente 2 Kbytes de memoria ROM (o flash), una vez compilado.

La memoria que necesita es muy poca además de la requerida por cada hilo. Como regla general, una aplicación pequeña requiere de 128 a 256 bytes de memoria, por hilo.

Se implementó para completar el sistema, un shell (Linea de comandos), una pila TCP/IP basada en uIP [Dunkels:2003] (Aunque re-escrita completamente) y un sistema de archivos, que aumentó la cantidad de líneas de código a 7000 aproximadamente, ocupando unos 30 Kbytes de memoria ROM. Los comentarios y variables del código fuente se realizaron en el idioma Ingles, para lograr una mayor difusión en caso de publicarse.

12. Conclusiones

Aunque el sistema carece actualmente de características que lo adecuan para el desarrollo de sistemas en tiempo real (latencia constante en el scheduler, soporte de diferentes ISR por parte del sistema operativo, etc.) en el futuro podrían llegar a implementarse estas características y otras, con poco trabajo.

El desarrollo del sistema operativo picix fue finalmente completado e implementado para la plataforma PIC-18, y como resultado se obtuvo un sistema con muchas posibilidades, ya que se facilita la programación rápida de sistemas complejos utilizando dispositivos microcontroladores de bajo precio. Los diseñadores de sistemas embebidos pueden encontrar interesante este tipo de software ya que permite generar aplicaciones modulares, en lugar del código monolítico utilizado generalmente.

[2] Como el microcontrolador ejecuta una instrucción cada exactamente 4 ciclos, cada cambio de contexto insume 700 instrucciones.

13. Glosario

ISR (Interrupt Service Routine) Rutina de servicio de interrupción, es llamada cada vez que se produce una interrupción de hardware.

PIC PIC surgió como una abreviatura de *Programmable Integrated Circuit*, pero hoy en día es una marca comercial, y no debe utilizarse como abreviatura. Se divide en familias: PIC12 en la gama baja, PIC16/PIC17 en la gama media y PIC18 en la gama alta.

CPU: (Central Process Unit, Unidad Central de Proceso) componente central de todo computador, encargado de ejecutar las instrucciones que se descargan de la memoria.

x86: Arquitectura de CPU desarrollada por Intel y muy utilizada en la actualidad para sistemas de escritorio o PCs.

Kernel: Kernel (Nucleo) es el conjunto de funciones pertenecientes al sistema operativo que tiene mayor control sobre el CPU que las funciones de usuario normales.

Referencias

[Sebesta:1996] Robert W. Sebesta (1996), **Concepts of Programming Languages**, Third Ed. , Addison-Wesley.

[Tanenbaum:1992] Andrew S. Tanenbaum (1992), **Modern Operating systems,** Prentice-Hall

[UYM:2000] Usategui, Yesa y Martínez (2000), **Microcontroladores PIC**, Diseño práctico de aplicaciones, McGraw-Hill

[Dunkels:2003] Dunkels, **Full TCP/IP for 8-Bit Architectures**, Swedish Institute of Computer Sciece, Mobisys 2003