ESCUELA POLITÉCNICA SUPERIOR DE CÓRDOBA
Universidad de Córdoba

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA Y DE COMPUTADORES

SISTEMAS EMPOTRADOS

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos

Profesor: Carlos Diego Moreno Moreno

Dpto. de Ingeniería Electrónica y de Computadores.
Área de Arquitectura y Tecnología de Computadores.
Escuela Politécnica Superior. Universidad de Córdoba.

UNIVERSIDAD D CÓRDOBA

Objetivos.

- Estudio de los sistemas operativos más sencillos que se emplean actualmente en los sistemas empotrados.
- Veremos solamente las técnicas de planificación cíclica.
- Estudiar dos planificadores cíclicos sencillos.
- Finalmente profundizaremos en el problema de la exclusión mutua en procesos y las soluciones posibles.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.1.– Introducción.

- En esta asignatura se van a considerar los sistemas operativos más sencillos que se utilizan habitualmente en los microcontroladores.
- Sistema operativo sencillo o *kernel*:
  - Reparte el tiempo de CPU entre las tareas: Administrador de tareas.
  - Abstracción del *hardware*.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.2.– Estrategias de planificación de tareas.

- Existen diversas estrategias de planificación de tareas, pero todas se pueden englobar en alguno de los siguientes grupos básicos:
  - Estrategias cíclicas.
  - Estrategias expropiativas o apropiativas.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.3.– Estrategias de planificación cíclica.

- La estrategia cíclica consiste en asignar las tareas a la CPU por turno, no permitiéndose la interrupción de una tarea antes de su finalización. Una tarea utiliza la CPU tanto tiempo como desea. Cuando no necesita la CPU, el planificador la asigna a la siguiente tarea de la lista:
  - **Ventaja** : Es la estrategia más simple y eficiente, ya que minimiza el tiempo de conmutación de tareas. Es una estrategia efectiva para pequeños sistemas empotrados para los que el tiempo de ejecución de cada tarea está cuidadosamente calculado y el *software* se ha dividido dentro de apropiadas tareas.
  - **Inconveniente**: Esta estrategia es demasiado restrictiva puesto que requiere que cada tarea tenga tiempos similares de ejecución y con ella es difícil hacer frente a los eventos aleatorios.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.4.– Planificadores cíclicos sencillos.

- Las estrategias de planificación cíclicas son las siguientes:
  - Estrategia basada en el ciclo menor y mayor, aplicable principalmente a tareas periódicas.
  - Estrategia "*Instant Up*", aplicable solamente a tareas periódicas.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.4.1.– Planificador basado en el ciclo menor y mayor.

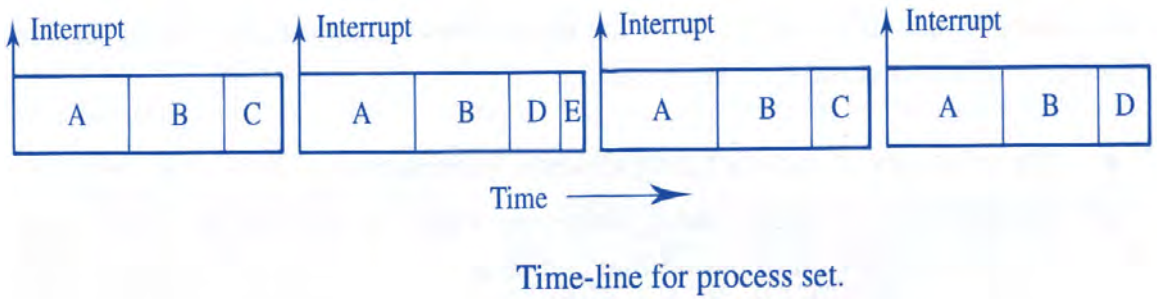
- Aplicable principalmente a tareas periódicas.
- La planificación se realiza estudiando el ciclo menor, que es el periodo menor escogido entre los periodos de las tareas, y el ciclo mayor, que es el periodo mayor escogido entre los períodos de las tareas.
- Por tanto, el ciclo mayor es el máximo tiempo en el que se tienen que haber ejecutado al menos una vez todas las tareas.

9.4.1.– Planificador basado en el ciclo menor y mayor.

- Ejemplo de planificación cíclica basada en el estudio del ciclo mayor (100ms) y menor (25ms).
- El conjunto de procesos se podría planificar en un ejecutivo cíclico como muestra el *Time–Line* de la figura. Cada proceso se asimilaría a una función que el ejecutivo irá llamando. Se tendría un ciclo principal de 100 ms, que es el tiempo en el que deben ejecutarse todos los procesos, y cuatro ciclos secundarios de 25 ms, equivalente al proceso que tiene un período menor.

Cyclic executive process set.

Process	*Period, T*	*Computation Time, C*
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2



Time-line for process set.

9.4.1.– Planificador basado en el ciclo menor y mayor.

- El programa principal se podría realizar con un lazo de la siguiente forma:

```
Loop
        Wait_for_interrupt;
                Procedure_For_A;
                Procedure_For_B;
                Procedure_For_C;
        Wait_for_interrupt;
                Procedure_For_A;
                Procedure_For_B;
                Procedure_For_D;
                Procedure_For_E;
        Wait_for_interrupt;
                Procedure_For_A;
                Procedure_For_B;
                Procedure_For_C;
        Wait_for_interrupt;
                Procedure_For_A;
                Procedure_For_B;
                Procedure_For_D;
End loop;
```

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.4.2.– Planificador "*Instant–Up*".

- Aplicable solamente a tareas periódicas.
- Para que un conjunto de tareas sea planificable mediante esta estrategia, se debe cumplir que la suma de los tiempos de ejecución de todas las tareas sea menor que el periodo de un *tick* (periodo de repetición de una tarea).
- A cada tarea se le asocia un temporizador, una vez que este temporizador llega al final, entonces la tarea se pone como lista para ejecutarse. El programa principal comprueba por orden las tareas que están listas para ejecutarse y las va ejecutando.
- Cada unidad de cuenta del temporizador corresponde con un *tick*.
- Los temporizadores se crean por programa. Estos temporizadores *software* se asocian a un temporizador *hardware*. Cada interrupción producida por el temporizador *hardware* decrementa la cuenta de los temporizadores *software*. Cuando un temporizador *software* asociado a la tarea llega a cero, esta tarea se pone lista para ejecución.

9.4.2.– Planificador "*Instant–Up*".

Temporizadores asociados a las tareas.

Definición de *timers software*:

```
struct
        {
        UINT32 (* pTareas)();      // Código asociado a la tarea
        UINT32 Delay;              // cuenta actual del temporizador
        UINT32 Period;             // periodo de ejecución de la tarea
        UINT32 RunMe;              // estado de la tarea: lista para ejecutarse
        } TAREA;
```

9.4.2.– Planificador "*Instant–Up*".

Creación de una tarea. Funciones del S.O.

```
unsigned int SOAgrego_Tarea(unsigned int (* pFuncion)(),const unsigned int DELAY,const unsigned int PERIOD)
{
	unsigned int Indice = 0;
	while ((SOTareas[Indice].pTareas!= 0) && (Indice < SOMAX_TAREAS))	// Rastrea si hay tareas en la lista
	{
		Indice++;
	}
	if (Indice == SOMAX_TAREAS)		// ¿Encontró el final de la lista?
	{
		printf("Error");
	}
	SOTareas[Indice].pTareas = pFuncion;
	SOTareas[Indice].Delay = DELAY;
	SOTareas[Indice].Period = PERIOD;
	SOTareas[Indice].RunMe = 0;
	return Indice;			// Retorna la posición de la tarea, para permitir luego su ubicación
}
```

9.4.2.– Planificador "*Instant–Up*".

Eliminación de una tarea.

```
void SOElimina_Tarea (unsigned int i)
{
    SOTareas[i].pTareas= 0;
    SOTareas[i].Delay = 0;
    SOTareas[i].Period =0;
    SOTareas[i].RunMe = 0;
}
```

9.4.2.– Planificador "*Instant–Up*".

Interrupción de tiempo real y actualización de los temporizadores asociados a las tareas.

```
__irq void T1_MANEJADOR_IRQ (void)
{
        unsigned int Indice;                                    // NOTA: Los cálculos son en *TICKS* (no en milisegundos)
        for (Indice = 0; Indice < SOMAX_TAREAS; Indice++)
        {
                if (SOTareas[Indice].pTareas)                   // Chequeo si hay tareas en esa posición
                {
                        if (SOTareas[Indice].Delay == 0)        // La tarea esta lista para ser ejecutada
                        {
                                SOTareas[Indice].RunMe += 1;    // Incremento la bandera 'RunMe'
                                if (SOTareas[Indice].Period)
                                {
                                                                // Organizo tarea periodica para ser ejecutada nuevamente cuando corresponda
                                        SOTareas[Indice].Delay = SOTareas[Indice].Period;
                                }
                        }
                        else
                        {
                                SOTareas[Indice].Delay -= 1;    // Tarea aun no lista para ser ejecutada: Solo decremento el Delay
                        }
                }
        }
        T1IR = 1;                                               // Limpio bandera de interrupción
        VICVectAddr = (unsigned int)0x0;                        // Reconocimiento de interrupción
```

9.4.2.– Planificador "*Instant–Up*".

Planificador/Despachador de tareas.

```
void  SODespachador_Tareas(void)
{
        unsigned int Indice;
        for (Indice = 0; Indice < SOMAX_TAREAS; Indice++)        // El despachador ejecuta la próxima tarea si es que hay una
        {
                if (SOTareas[Indice].RunMe > 0)
                {
                        (*SOTareas[Indice].pTareas)();           // Ejecuta la tarea
                        SOTareas[Indice].RunMe -= 1;             // Bajo la bandera RunMe
                        if (SOTareas[Indice].Period == 0)        // Si la tarea no es periódica, se ejecuta solo una vez y se elimina de la lista
                        {
                                SOEliminaTarea(Indice);
                        }
                }
        }
}
```

9.4.2.– Planificador "*Instant–Up*".

Integración en una aplicación.

- El sistema operativo se integra como parte de una nuestra aplicación. Por tanto, además de disponer del código fuente, deben hacerse una serie de inicializaciones y llamadas a funciones de inicialización del sistema operativo.

9.4.2.– Planificador "*Instant–Up*".

Integración en una aplicación.

- Los pasos a seguir en la función *main* de nuestra aplicación son:
  - Se deben inicializar las variables globales del S.O. Se debe llamar a la función:
    - SO_Inicio_TCB_Var()
  - Se inicializa el temporizador *hardware* asociado al S.O. (*timer 1* en el código original) habilitando la interrupción y arrancando la cuenta:
    - ActVIC_T1() // coloca los valores en el VIC para el T1
    - startT1() // timer asociado al SO
  - Una vez inicializados el resto de los periféricos, se arranca el S.O. habilitando la interrupción de tiempo real (en el código original la ISR asociada al *timer* 1):
    - SOStart()
  - Se definen las tareas al S.O. Haciendo una llamada por tarea a la función:
    - SOAgrego_Tarea(pointerTask,delay,period);
  - En la última línea de la función *main* se llama continuamente al planificador/despachador:
    - while(1)SODespachador_Tareas();

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.5.– Exclusión mutua en Sistemas Operativos.

OBSERVACIÓN MUY IMPORTANTE.

- En el apartado de exclusión mutua, debe tenerse en cuenta cuando se habla de un sistema operativo general y cuando se está concretando a un sistema operativo sencillo.
- De igual forma, se debe tener en cuenta, si estamos refiriéndonos a un sistema monoprocesador o multiprocesador, ya que la validez de las afirmaciones que se hagan depende de cada caso.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.5.1.– Secciones críticas. Planteamiento del problema.

- Supongamos por ejemplo *X=X+1;*
- Esta sentencia de un lenguaje de alto nivel cualquiera, por ejemplo C, no se ejecutará como una operación indivisible sino en tres micro–operaciones:
  - Se carga el valor de *X* (dirección de memoria) en un registro.
  - Se incrementa en *1* el valor del registro.
  - Se lleva el valor del registro a *X* (dirección de memoria).
- Si ahora los procesos se entremezclan y quieren actualizar la variable *X* se producirá un valor incorrecto.

9.5.1.– Secciones críticas. Planteamiento del problema.

- Por ejemplo, si ocurriera la secuencia:
  - X=5 al inicio
  - El proceso P1 carga el valor 5
  - Se cambia la planificación y entra el proceso P2
  - El proceso P2 carga el valor 5
  - Incrementa el proceso P2
  - Lleva el valor del registro, 6 a X
  - En otro momento se devuelve el control al proceso P1
  - Incrementa el proceso P1
  - Lleva el valor del registro, 6 a X.
- Al final X valdrá 6 y no 7 como debería ser. Por tanto, se tiene un resultado incorrecto.

9.5.1.– Secciones críticas. Planteamiento del problema.

- Un conjunto de operaciones que deberían ejecutarse indivisiblemente se le llama **sección crítica**.
- O lo que es lo mismo, aquella parte del programa que accede a memoria compartida se le llama **sección crítica** o región crítica (algunos autores diferencian entre sección crítica y región crítica).
- La sincronización requerida para proteger una sección crítica se le llama **exclusión mutua**.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.5.2.– Posibles soluciones:
9.5.2.1.– Inhabilitación de interrupciones.

- Una solución al problema de la exclusión mutua es que un proceso inhabilite las interrupciones justo al entrar en su sección crítica y las habilite justo al salir. Con esto se garantiza que no se conmutará a otro proceso.
- El defecto de este método es que es arriesgado conferir a los procesos de usuario la posibilidad de actuar sobre las interrupciones porque podría este proceso no habilitarlas y el sistema se quedaría colgado.
- Por otro lado, si el sistema es multiprocesador la inhabilitación de interrupciones sólo afectaría a los procesos que estuvieran en ese procesador y no los del resto.
- La utilización de las interrupciones es una técnica útil dentro del sistema operativo, pero no es apropiada como mecanismo de exclusión mutua general para los procesos de usuario.
- En sistemas operativos cíclicos y suponiendo un sistema monoprocesador es la técnica más útil, ya que todos los procesos, incluidos los de usuario, son procesos de sistema.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.5.2.– Posibles soluciones:
9.5.2.2.– Espera activa (*Busy Waiting*):

- En este apartado vamos a ver las soluciones y los defectos de las propuestas para la exclusión mutua con espera activa, es decir, mientras un proceso está en su región crítica el otro espera ejecutando un ciclo de espera.
- Los mecanismos que vamos a estudiar son:
  - Variables candado (NO consigue la exclusión mutua).
  - Alternancia estricta.
  - Solución de *Peterson*.
  - Instrucción TSL (*Test and Set Lock*).

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.5.2.– Posibles soluciones:
9.5.2.2.– Espera activa (*Busy Waiting*): Variables candado.

- Se trata de una solución *software*. Se define una variable compartida que indique si un proceso se encuentra o no, en su sección crítica. Supongamos una variable candado cuyo valor inicial sea por ejemplo cero. Un proceso que quiera entrar en su sección crítica realiza los siguientes pasos:
  - Comprueba si el candado está a 1, si no es así, el proceso le asigna a esta variable 1 y entra en su sección crítica.
  - Si el candado está a 1 espera a que esté a 0.
- **El problema que tiene este método es que NO resuelve realmente el problema de la exclusión mutua. Se muestra habitualmente para demostrar que esta solución intuitiva, NO es efectiva**.

9.5.2.– Posibles soluciones:

9.5.2.2.– Espera activa (*Busy Waiting*): Variables candado.

- En el siguiente ejemplo se ha considerado dos procesos y dos variables candado: *flag1* y *flag2*.

```
process P1;
  loop
    flag1 := up; (* announce intent to enter *)
    while flag2 = up do
      null     (* busy wait if the other process is in *)
    end;       (* its critical section *)
    <critical section>
    flag1 := down; (* exit protocol *)
    <non-critical section>
  end
end P1;
```

```
process P2;
  loop
    flag2 := up;
    while flag1 = up do
      null
    end;
    <critical section>
    flag2 := down;
    <non-critical section>
  end
end P2;
```

Both processes announce their intention to enter their critical sections and then check to see if the other process is in its critical section. Unfortunately, this 'solution' suffers from a not insignificant problem. Consider an interleaving that has the following progression:

```
P1 sets its flag (flag1 now up)
P2 sets its flag (flag2 now up)
P2 checks flag1 (it is up therefore P2 loops)
P2 enters its busy wait
P1 checks flag2 (it is up therefore P1 loops)
P1 enters its busy wait
```

9.5.2.– Posibles soluciones:
9.5.2.2.– Espera activa (*Busy Waiting*): Variables candado.

- En el siguiente ejemplo se ha considerado dos procesos y dos variables candado: *flag1* y *flag2* y se ha cambiado el orden en la activación y comprobación de los candados respecto al caso anterior.

```
process P1;
  loop
    while flag2 = up do
      null (* busy wait if the other process is in *)
    end;    (* its critical section *)
    flag1 := up; (* announce intent to enter *)
    <critical section>
    flag1 := down; (* exit protocol *)
    <non-critical section>
  end
end P1;
```

```
process P2;
  loop
    while flag1 = up do
      null
    end;
    flag2 := up;
    <critical section>
    flag2 := down;
    <non-critical section>
  end
end P2;
```

Now we can produce an interleaving that actually fails to give mutual exclusion.

```
P1 and P2 are in their non-critical sections (flag1 = flag2 = down)
P1 checks flag2 (it is down)
P2 checks flag1 (it is down)
P2 sets its flag (flag2 now up)
P2 enters critical section
P1 sets it flag (flag1 now up)
P1 enters critical section
(P1 and P2 are both in their critical sections).
```

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.5.2.– Posibles soluciones:
9.5.2.2.– Espera activa (*Busy Waiting*): Alternancia estricta.

- Consiste en definir un testigo, y los procesos solamente pueden entrar en la sección critica si tienen ese testigo. Veamos un ejemplo en C:

```
while (TRUE) {
  while(turn != 0) /* esperar */ ;
  critical_region();
  turn = 1;
  noncritical_region();
}
```

(a)

```
while (TRUE) {
  while(turn != 1) /* esperar */ ;
  critical_region();
  turn = 0;
  noncritical_region();
}
```

(b)

- La estrategia de alternancia no es una buena solución ya que, aunque soluciona el problema de la exclusión mutua, un proceso puede bloquear al otro aunque no se encuentre en su sección crítica.
- Principalmente, el problema de la alternancia estricta es que, si un proceso es mucho más lento que otro, obliga a que los dos progresen a la velocidad del más lento, se encuentre o no ejecutando una sección crítica.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.5.2.– Posibles soluciones:
9.5.2.2.– Espera activa (*Busy Waiting*): Solución de Peterson.

- En 1981 *G. L. Peterson* dio una solución *software* para resolver el problema de la exclusión mutua mejorando otras soluciones *software* anteriores.
- El algoritmo de Peterson consta de dos procedimientos:
  - Enter_region(número de proceso).
  - Leave_region(número de proceso).
- Antes de utilizar las variables compartidas, cada proceso debe invocar *enter_región* con su número de proceso como parámetro. Esta invocación le obligará a esperar hasta que pueda entrar sin peligro y manipular las variables compartidas. Para indicar que se ha finalizado, se invoca la función *leave_region*, permitiendo así que otro proceso pueda entrar.

9.5.2.– Posibles soluciones:
9.5.2.2.– Espera activa (*Busy Waiting*): Solución de Peterson.

```
#define FALSE       0
#define TRUE        1
#define N           2                   /* número de procesos */

int turn;                               /* ¿a quién le toca? */
int interested[N];                      /* todos los valores son inicialmente 0 (FALSE) */

void enter_region(int process);         /* proceso 0 o 1 */
{
 int other;                             /* número del otro proceso */

 other = 1 – process;                   /* lo opuesto de process */
 interested[process] = TRUE;            /* mostrar interés */
 turn = process;                        /* establecer bandera */
 while (turn == process && interested[other] == TRUE) /* instrucción nula */ ;
}

void leave_region(int process)          /* process: quién sale */
{
 interested[process] = FALSE;           /* indicar salida de la región crítica */
}
```

Solución de Peterson para lograr la exclusión mutua.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.5.2.– Posibles soluciones:
9.5.2.2.– Espera activa (*Busy Waiting*): Instrucción TSL (*Test and Set Lock*).

- La instrucción **TSL** (***T**est and **S**et **L**ock*, probar y fijar candado), lee el contenido de una posición de memoria, lo coloca en un registro y almacena un valor distinto de cero.
- Se garantiza que las operaciones de escritura y lectura son indivisibles.
- Son instrucciones que deben estar incluidas en el repertorio de instrucciones del procesador.
- Imprescindible en entornos multiprocesador, ya que es la única solución válida.
- En las implementaciones reales este tipo de instrucción varía en cada procesador y el funcionamiento no es el mismo, pero la idea es similar.

9.5.2.– Posibles soluciones:
9.5.2.2.– Espera activa (*Busy Waiting*): Instrucción TSL (*Test and Set Lock*).

- A continuación se muestra como se utiliza la instrucción TSL para implementar las funciones *enter_region* y *leave_region*:

```
enter_region:
        tsl register,lock       | copiar lock en register y asignarle 1
        cmp register,#0         | ¿era lock 0?
        jne enter_region        | si no era cero, se asignó 1 a lock, y se ejecuta el ciclo
        ret                     | volver al invocador; se entró en la región crítica

leave_region:
        move lock,#0            | guardar un 0 en lock
        ret                     | volver al invocador
```

Establecimiento y liberación de candados con TSL.

9.5.2.– Posibles soluciones:
9.5.2.2.– Espera activa (*Busy Waiting*): Inconvenientes.

- La solución de *Peterson* y la que utiliza TSL son correctas, pero tienen el defecto de requerir espera activa.
- Esta espera activa, presenta dos graves inconvenientes:
  - Se consume tiempo de CPU.
  - Se puede presentar el problema de **inversión de prioridad**.
- **Inversión de prioridad:** Ejemplo: Supongamos dos procesos **H** y **L**. **H** tiene más prioridad que **L**.
  - **L** entra en su región crítica.
  - **H** queda listo para ejecutarse
  - Se produce un cambio de planificación y pasa a ejecutarse **H**.
  - **H** inicia la espera activa.
  - Como **L** nunca se planifica, por tener menor prioridad que **H**, nunca saldrá de su región crítica y **H** permanece en un lazo infinito.

Tema 9 – Diseño de sistemas operativos sencillos: Sistemas operativos cíclicos.

9.5.3.– Exclusión mutua en sistemas operativos cíclicos y sistemas monoprocesador.

- Aunque el sistema operativo sea cíclico, y una tarea no pueda interrumpir a otra hasta que no haya finalizado, si se utilizan interrupciones, puede haber variables compartidas entre las ISR y las tareas. Por tanto, se tendrá que utilizar algún mecanismo de exclusión mutua entre las ISR y las tareas que compartan variables.
- En el caso de sistemas operativos cíclicos, tanto las tareas de usuario como el código asociado al sistema operativo, tienen las mismas características, todas son «tareas y funciones de sistema» y por tanto, los inconvenientes de inhabilitar las interrupciones comentados al comienzo del tema para conseguir la exclusión mutua, no se aplican.
- En consecuencia, si además estamos en un sistema monoprocesador, la única solución para la exclusión mutua es:

LA INHABILITACIÓN DE INTERRUPCIONES

9.5.3.– Exclusión mutua en sistemas operativos cíclicos y sistemas monoprocesador.

- De todos los métodos de espera activa comentados en el caso de sistemas operativos cíclicos, no es necesario que se aplique ninguno, si las variables están compartidas entre las tareas y no aparecen en las ISR.
- Resumiendo, para el caso de sistemas monoprocesador:
  - Si las variables están compartidas únicamente entre las tareas, no es necesaria la exclusión mutua.
  - Si las variables están compartidas entre tareas e ISR, se utilizará la inhabilitación de interrupciones, cuando las tareas manipulen las variables compartidas y se habilitarán al finalizar.
  - Del lado de las ISR, si el sistema de interrupciones no es multinivel, no es necesario utilizar ningún mecanismo ya que las ISR nunca serán interrumpidas.
  - De todos los mecanismos comentados para exclusión mutua en este tema solamente es efectivo: **la inhabilitación de interrupciones.**

Bibliografía consultada.

- Tanenbaum, Andrew S. “Sistemas operativos: diseño e implementación”. Editorial Prentice Hall. 1998. 2ª edición. ISBN : 970–17–0195–8.

¡Muchas gracias por su atención!

Carlos Diego Moreno Moreno

Área de Arquitectura y Tecnología de Computadores
Departamento de Ingeniería Electrónica y Computadores.
Escuela Politécnica Superior. Universidad de Córdoba