

ESCUELA SUPERIOR POLITECNICA DEL LITORAL
FACULTAD DE INGENIERIA ELECTRICA

**" INVESTIGACION E IMPLANTACION DE UN
SISTEMA DE MULTIPROGRAMACION USANDO
LENGUAJE MODULA-2 "**

TESIS DE GRADO

**Previa a la Obtención del Título de:
INGENIERO EN ELECTRICIDAD**

Especialización: ELECTRONICA

Presentada por:

Juan Carlos Moncayo Cárdenas

Guayaquil - Ecuador

1989

AGRADECIMIENTO

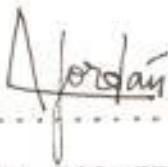
Al Ing. Jaime Puente por su
valiosa ayuda en la realización
de este trabajo.



ING. CARLOS VILLAFUERTE P.
Sub-Decano de la Facultad
Ingeniería Eléctrica



ING. JAIME PUENTE P.
Director de Tesis



ING. CARLOS JORDÁN
Miembro Principal



ING. JAIME SANTORO
Miembro Principal

DECLARACION EXPRESA

"La responsabilidad por los hechos, ideas y doctrinas expuestas en esta tesis, me corresponden exclusivamente; y, el patrimonio intelectual de la misma, a la ESCUELA SUPERIOR POLITECNICA DEL LITORAL"

(Reglamento de Exámenes y Títulos Profesionales de la ESPOLY)

.....
JUAN CARLOS MONCAYO CARDENAS

RESUMEN

El presente trabajo introduce el tema de programación concurrente y presenta una implantación usando el lenguaje Modula-2.

El capítulo I aborda los conceptos básicos necesarios para entender qué es programación concurrente, cómo implantarla, cuáles son las herramientas requeridas para esta implantación y además presenta tres problemas clásicos que ilustran la naturaleza de programación concurrente. Todos los conceptos presentados en este capítulo, tales como: procesos, concurrencia, cuasi-concurrencia, exclusión mutua, semáforos y otros; sirven de base teórica para una implantación de los mismos en el capítulo III y IV, utilizando Modula-2. La implantación que realizaremos se basa en el concepto de cuasi-concurrencia y se hará en un microcomputador IBM PC basado en el microprocesador 8088 de INTEL.

El capítulo II es una introducción a Modula-2; aquí se investiga las propiedades de este lenguaje que posibilitan trabajar en programación concurrente. En particular se analiza el módulo SYSTEM , el cual

provee algunos tipos de datos y procedimientos que permiten operar con procesos concurrentes. Además se presenta el módulo PROCESSEG que provee una abstracción de comunicación entre procesos y que soporta cuasi-concurrencia.

En el capítulo III se presentan tres problemas tradicionales dentro de programación concurrente. Estos problemas se resuelven tomando como base los conceptos analizados en el capítulo I y utilizando las herramientas suministradas por Modula-2. Estos problemas son: productor/consumidor, lectores/escritores y problema de los filósofos.

El capítulo IV ilustra una implantación de cuasi-concurrencia diferente a la presentada en el capítulo III, en el que la asignación del procesador corre a cargo de los procesos. Ahora la asignación del procesador la lleva a cabo un módulo manejador o planificador de tareas, el mismo que asigna a cada proceso un tiempo fijo para uso del procesador, al término del cual el control retorna al manejador de procesos y éste a su vez asigna el procesador a otro proceso y así sucesivamente.

El trabajo consta además de tres apéndices que sirven como referencia adicional .

INDICE GENERAL

PAG.

RESUMEN	VI
INDICE GENERAL	VIII
INDICE DE FIGURAS	XI
INTRODUCCION	12
CAPITULO I	14
PROCESOS CONCURRENTES	14
1.1 Introducción	14
1.2 Definiciones	14
1.2.1 Procesos	15
1.2.2 Concurrencia	17
1.3 Notación para Programación Concurrente	19
1.4 Exclusión Mutua	19
1.4.1 Introducción	19
1.4.2 Recursos Compartidos	20
1.4.3 Datos Compartidos	23
1.5 Interacción entre Procesos	24
1.5.1 Comunicación entre Procesos	24
1.5.2 Semáforos	26
1.5.3 Secciones Críticas	28
1.5.4 Monitores	32
1.6 Bloqueo de Procesos	35
1.7 Algoritmos de Programación Concurrente	36
1.7.1 Problema Productor-Consumidor	36
1.7.2 Problema Lectores y Escritores	37
1.7.3 Problema de los Filósofos	37

CAPITULO II	39
INTRODUCCION A MODULA-2	39
2.1 Introducción	39
2.2 Módulos	39
2.2.1 Definición e Implementación	42
2.3 Entrada y Salida Secuencial	45
2.4 Procesos en Modula-2	47
2.4.1 Corrutinas	47
2.4.2 El Tipo PROCESS	48
2.4.3 El Procedimiento NEWPROCESS	49
2.4.4 El Procedimiento TRANSFER	49
2.4.5 El Módulo PROCESSES	51
2.5 Facilidades de Bajo Nivel	59
2.6 Manejo de Dispositivos e Interrupciones	63
CAPITULO III	66
IMPLANTACION DE ALGORITMOS USANDO MODULA-2	66
3.1 Problema Productor-Consumidor	66
3.2 Problema Lectores y Escritores	89
3.3 Problema de los Filósofos	106
CAPITULO IV	
ILUSTRACION DE PROGRAMACION CONCURRENTE Y COMUNICACION ENTRE PROCESOS	116
4.1 Introducción	116
4.2 Algoritmo 'Round-Robin'	118
4.3 Consideraciones De Tiempo	120

4.4 Módulo Manejador de Procesos	123
4.4.1 Módulos de Definición e Instalación	123
4.4.2 Descripción de los Módulos	135
4.5 Demostración del Manejador de Procesos	143
4.6 Observaciones Generales	148
CONCLUSIONES Y RECOMENDACIONES	151
APENDICES	159
APENDICE A. Manual del Usuario	160
APENDICE B. Referencias Técnicas del IBM PC	164
APENDICE C. Módulos de Biblioteca Utilizados	166
BIBLIOGRAFIA	171

INDICE DE FIGURAS

#	PAG.
2.1 Sintaxis de un módulo	40
2.2 Sintaxis de listas de importación y exportación	41
2.3 Sintaxis de un módulo de definición	43
2.4 Descriptor de Proceso	57
2.5 Sintaxis de una especificación de memoria absoluta..	64
3.1 Problema de los Filósofos.....	107
4.1 Estados de un proceso	117
4.2 Algoritmo 'Round-Robin'	119
B.1 Diagrama de bloques del INTEL 8253	165

INTRODUCCION

Una de las partes más importantes en los sistemas de computadores es el sistema operativo, el cual es un conjunto de programas que controlan el funcionamiento del computador. Entre sus tareas está la asignación de recursos a los diferentes usuarios del computador; estos recursos pueden ser : memoria, dispositivos de entrada/salida, tiempo de procesamiento y otros. Para hacer estas tareas los sistemas operativos se basan en los conceptos de programación concurrente. Desde este punto de vista, el presente trabajo tiene como objetivo ayudar a comprender el funcionamiento de un sistema operativo, ya que aborda conceptos que son básicos dentro de este campo, tales como: procesos, semáforos, monitores, entre otros, además de que presenta un manejador de procesos, el cual se encarga de la asignación del procesador entre los procesos.

Otro punto importante es el concepto de lenguaje de alto nivel que incluye aspectos como: modularidad o capacidad para dividir un programa en varios subprogramas, facilidades para programación estructurada, posibilidad de usar este tipo de lenguajes en el diseño de sistemas operativos,etc.. En este sentido, Modula-2 tiene cualidades que lo hacen apropiado para muchas

aplicaciones; entre estas cualidades podemos mencionar el concepto de módulos, compilación separada, exportación e importación de objetos, facilidad para soportar concurrencia de procesos e interaccionar con interrupciones externas, etc.. Estos conceptos son investigados en el presente trabajo.

Además consideramos que es importante analizar las facilidades que ofrecen diferentes lenguajes para implantación de software de control de procesos, para lo cual esta tesis hace un estudio de las posibilidades que ofrece Modula-2 para este tipo de software y presenta una implementación elemental.

Cabe indicar que lo que se ha hecho en este trabajo puede ser un paso inicial para posteriores implantaciones basadas en Modula-2.

CAPITULO I PROCESOS CONCURRENTES

1.1 INTRODUCCION

Este capítulo introduce el tema de procesos concurrentes. Aquí se presentarán los conceptos básicos para luego desarrollar e implantar una aplicación de programación concurrente basada en el lenguaje Módula-2, el cual será introducido en el siguiente capítulo.

Aquí se describen los problemas de exclusión mutua, regiones críticas, comunicación entre procesos, monitores, etc. y se sigue una nomenclatura generalmente adoptada en textos sobre programación concurrente y sistemas operativos; en particular la nomenclatura utilizada en este capítulo es la de Per Brinch Hansen (1).

Además en la última parte (sección 1.7) se presentan tres problemas que son clásicos en el estudio de programación concurrente y que serán resueltos utilizando Módula-2 en el capítulo III.

1.2 DEFINICIONES

En esta parte se dan algunas definiciones básicas para

programación concurrente, como son: datos, operaciones, procesos.

El término DATOS se puede definir como unidades de información que son manipuladas por un proceso .

Las reglas de manipulación de datos se definen como OPERACIONES. Un sistema operativo se puede considerar como un conjunto de programas que tienen como objetivo administrar los recursos de un computador entre los diferentes procesos que hacen uso de ellos. Estos recursos pueden ser: memoria, dispositivos de entrada y salida de datos, procesador, etc.

1.2.1 PROCESOS

Un PROCESO es una secuencia de operaciones realizadas una a la vez. Durante su existencia, un proceso pasa por una serie de estados discretos. Varios eventos pueden causar la transición de un estado a otro. Si un proceso en cierto momento tiene el control de la Unidad Central de Proceso (UCP), se dice que su estado es de 'carrera', un proceso está en estado 'listo' cuando está en condiciones de utilizar la UCP (ocupada en ese momento por otro proceso), un proceso está 'bloqueado' si está esperando la ocurrencia de un evento, que le

permitirá posteriormente convertirse en listo; existen también otros estados posibles.

Un proceso en un sistema operativo se puede representar con lo que se denomina 'Bloque de Control de Proceso' ('Process Control Block' o PCB). El PCB es una estructura de datos que contiene información importante acerca del proceso, que incluye:

- Estado actual del proceso
- Identificación única del proceso
- Prioridad del proceso
- Punteros que localizan la posición en memoria del proceso
- Punteros para asignar recursos
- Una área de 'pila'

El PCB es un registro de información que permite al sistema operativo localizar toda la información necesaria sobre el proceso. Cuando la UCP es compartida por varios procesos (lo que veremos en el capítulo III y capítulo IV) usa el área de pila en el PCB donde mantiene la información necesaria para reanudar la ejecución de un proceso, cuando éste ha obtenido el control de la UCP.

Un PCB entonces mantiene la información que representa o define a un proceso ante el sistema operativo.

Sistemas que manejan procesos, son capaces de realizar ciertas operaciones sobre ellos, tales como:

- Crear procesos
- Destruir procesos
- Suspender un proceso
- Reiniciar un proceso
- Cambiar la prioridad de un proceso
- Bloquear un proceso
- Despertar un proceso

A su vez, la creación de un proceso implica varias operaciones como:

- Dar un nombre al proceso
- Insertar en la lista de procesos conocidos
- Determinar la prioridad inicial del proceso
- Crear el PCB
- Asignar los recursos iniciales del proceso

Estos conceptos utilizaremos en los capítulos III y IV. En el capítulo II se da una definición de procesos desde el punto de vista de Modula-2.

1.2.2 CONCURRENCIA

Se dice que dos procesos son concurrentes si se ejecutan al mismo tiempo e interactúan entre ellos por medio de

un recurso compartido. Esa interacción puede ser en la forma de envío de señales para avisar a otro proceso de la ocurrencia de algún evento, intercambio de datos entre ellos por medio de secciones críticas, etc., todo lo cual se analizará en las secciones subsiguientes.

Se pueden considerar los siguientes tipos de concurrencia:

- 1) Se dice que existe concurrencia real cuando el computador consiste de varios procesadores (UCP) y cada proceso se ejecuta en uno de ellos.
- 2) Se dice que existe quasi-concurrencia cuando el computador consiste de un único procesador, que es asignado a cada proceso por un espacio de tiempo (es compartido en el tiempo). Los procesos son llamados quasi-concurrentes.

En el presente trabajo discutiremos la formulación de procesos y sus interacciones en términos de Módula-2, y nuestra implantación se basará en un computador de un único procesador (o microprocesador, en el caso de un computador personal) que es el IBM-PC y en el concepto de corrutina, es decir sistemas de quasi-concurrencia.

Para esto introduciremos el módulo PROCESSES, que presenta las facilidades necesarias para quasi-concurrencia (sección 2.4.5).

1.3 NOTACION PARA PROGRAMACION CONCURRENTE

La notación para programación concurrente usada en el presente capítulo pertenece al lenguaje Pascal Concurrente; esta notación es utilizada en textos sobre sistemas operativos (2).

La notación COBEGIN S1;S2;....;Sn COEND indica que S1,S2,...,Sn son ejecutados concurrentemente, REPEAT-FOREVER representa un lazo que se ejecuta indefinidamente, REPEAT-UNTIL 'condición' es un lazo que se ejecuta hasta que 'condición' es verdadera. La notación para semáforos, secciones críticas, etc. se verán a medida que se traten dichos puntos.

Además para la implantación de los diferentes programas en los siguientes capítulos se hará uso de varias estructuras y tipos de datos propios del lenguaje Modula-2 como son: las estructuras LOOP-END, WHILE, REPEAT, IF-THEN, y tipos de datos como el tipo arreglo (ARRAY), registro (RECORD), puntero (POINTER TO) y otros (2).

1.4 EXCLUSION MUTUA

1.4.1 INTRODUCCION

El concepto de exclusión mutua es importante en sistemas de multiprogramación. En estos sistemas varios procesos pueden compartir una misma variable, una misma área de memoria u otros recursos. Para esto se requiere una forma de sincronizar los procesos, cuando tienen que acceder al mismo recurso.

Exclusión mutua garantiza que solamente un proceso puede hacer uso de un recurso compartido (RC) en un momento dado, y que si uno o varios procesos requieren acceder a ese recurso, no puedan hacerlo y deban esperar que el proceso en uso del recurso lo libere, lo cual tiene que ocurrir dentro de un período de tiempo finito. De esta manera, cada proceso que accede a un RC excluye a los otros procesos de hacerlo al mismo tiempo, es decir hay exclusión mutua entre ellos.

1.4.2 RECURSOS COMPARTIDOS

En esta sección se presenta un ejemplo de dos procesos concurrentes que comparten un recurso, para lo cual se requiere exclusión mutua entre los dos procesos. El algoritmo que se presenta a continuación usa dos variables de tipo BOOLEAN que indican cuál proceso está usando el recurso en un instante dado:

```
VAR Pturno,Qturno:BOOLEAN;
```

```

BEGIN
    Pturno:=FALSE; Qturno:=FALSE;
COBEGIN
    'P' REPEAT
        Pturno:=TRUE;
        REPEAT UNTIL NOT Qturno;
        usa recurso;
        Pturno:=FALSE;
        P no está usando recurso;
    FOREVER
    'Q' REPEAT
        Q Turno:=TRUE;
        REPEAT UNTIL NOT Pturno
        usa recurso;
        Qturno:=FALSE;
        Q no está usando recurso;
    FOREVER
COEND
END.

```

La asignación de la variable Pturno es hecha solamente por el proceso P.

Cuando P no usa el recurso, Pturno es falso, lo mismo para Qturno y Q. Se observa que los dos procesos son simétricos y se puede concluir que:

P en uso del recurso 'implica' Q no en uso del recurso
Q en uso del recurso 'implica' P no en uso del recurso

Exclusión mutua está garantizada entre los procesos.
Pero si los procesos hacen las asignaciones:

```
Pturno:=TRUE; Oturno:=TRUE
```

al mismo tiempo (en el caso de concurrencia real)
entonces permanecerán en sus lazos
indefinidamente, esperando que Pturno y Oturno sean
falso.

Se produce entonces lo que se denomina un 'Bloqueo'
('Deadlock', sección 1.6). Con lo visto hasta aquí, se
puede apreciar el problema de exclusión mutua y señalar
los siguientes criterios para su solución:

- 1) El recurso en cuestión puede ser usado por un solo proceso a la vez.
- 2) Cuando el recurso es requerido por varios procesos al mismo tiempo, debe ser obtenido por uno de ellos en un tiempo finito.
- 3) Cuando un proceso tiene un recurso ,éste tiene que ser liberado dentro de un tiempo finito.

Cuando un proceso está ejecutando el lazo REPEAT-UNTIL está haciendo uso del procesador sin mayor provecho, lo deseable es que el proceso espere en algún lugar el momento de usar el recurso permitiendo así a otro proceso usar el procesador. En la sección 1.5.2 veremos una manera de trabajar bajo este criterio.

1.4.3. DATOS COMPARTIDOS

Varios procesos pueden producir y consumir datos en una misma área de memoria o en la misma variable. Este es un ejemplo de interacción entre procesos.

El acceso a esa variable por parte de cada proceso tiene que ser sincronizado, para mantener dentro de la variable los valores reales y evitar errores si un proceso toma el valor de la variable alterado por otro proceso.

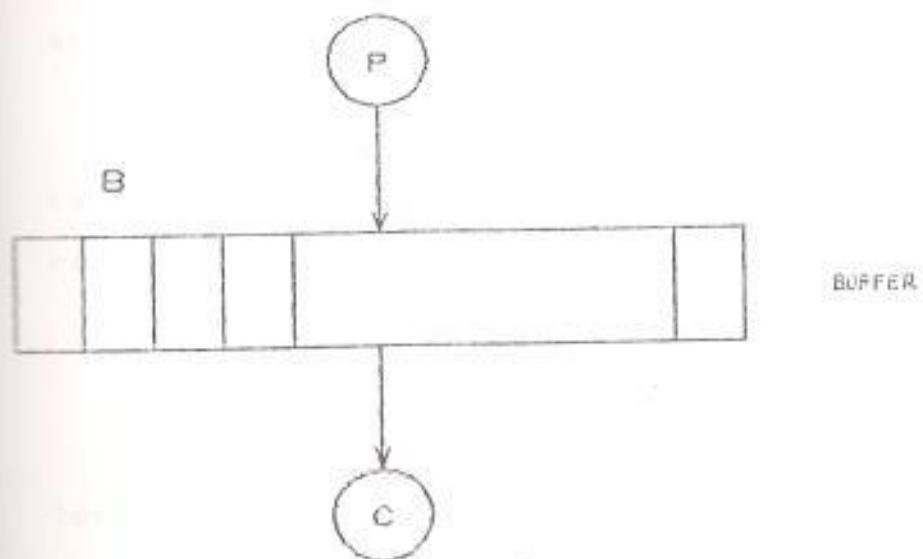
En las siguientes secciones se introducirán mecanismos de sincronización entre procesos tales como semáforos, regiones críticas, etc. y además se planteará el problema productor-consumidor que ilustra el manejo correcto de datos compartidos por varios procesos.

1.5 INTERACCION ENTRE PROCESOS

1.5.1 COMUNICACION ENTRE PROCESOS

La cooperación entre tareas, exige que los procesos puedan intercambiar datos. Un proceso P produce y envía datos a un proceso C, el cual recibe y consume dichos datos. Los datos son transmitidos entre los procesos en porciones discretas que se pueden llamar 'mensajes'.

Es posible que la velocidad de transmisión y recepción de datos sean diferentes, de tal manera que P envía datos más rápidamente de lo que C puede recibir, es por esto que se introduce una área de almacenamiento o acumulador de datos ('buffer') denominado B, como se muestra en la siguiente figura.



Observaciones importantes:

- 1) El proceso P no puede exceder la capacidad finita de B.
- 2) El proceso C no puede consumir datos más rápidamente de lo que son producidos.

Lo anterior conlleva a las siguientes reglas de comunicación: 1) si el acumulador está lleno y P trata de enviar datos a B, entonces P tiene que esperar hasta que C tome uno(más) dato(s) del acumulador, 2) si el acumulador está vacío y C trata de tomar datos de B, C debe esperar hasta que P envíe uno(más) dato(S) a B.

En la siguiente sección veremos la forma de implantar estas reglas de comunicación entre los procesos y en la sección 1.7.1 veremos un ejemplo que ilustra esta situación.

La secuencia de datos enviados y recibidos, puede ser representada conceptualmente como:

```
VAR S,R : ARRAY [0..∞] OF 'mensajes';
```

los mensajes son enviados y recibidos por medio de los siguientes procedimientos:

```
SEND(M,B)  
RECEIVE(M,B)
```

1.5.2 SEMAFOROS

En esta sección se considera la forma más simple de comunicación de procesos, el intercambio de señales de sincronismo. En algunos casos un proceso espera recibir solamente una señal de 'aviso' enviada por otro proceso cuando ha ocurrido un cierto evento, y aparte de esto no se requiere de otro intercambio de datos. Esto se puede considerar como un tipo especial de comunicación, en el cual un 'mensaje vacío' es enviado cada vez que un evento ha ocurrido, y así, es suficiente contar el número de estos mensajes.

Una variable tipo semáforo puede ser declarada como:

```
VAR s :SEMAPHORE;
```

Las operaciones 'Send' y 'Receive' anteriores, pueden ser llamadas:

```
SEND(s) y WAIT(s)
```

que originalmente fueron llamadas V y P respectivamente por Dijkstra. Ya que un semáforo es una variable común para los procesos P y C, las operaciones SEND y WAIT

sobre el mismo semáforo deben excluirse en el tiempo.

Un Semáforo s es una variable entera, la cual puede tomar solamente valores mayores o iguales que cero. Una vez que a s se asigna un valor inicial, mediante una operación de inicialización del semáforo, las únicas operaciones permitidas en adelante sobre s son SEND Y WAIT. Estas operaciones se definen como sigue:

WAIT(s): Si $s > 0$, entonces $s := s - 1$ y la ejecución del proceso que llamó a WAIT(s) continua; si $s = 0$ el proceso es suspendido.

SEND(s): Si existen procesos esperando por s, uno de ellos reinicia su ejecución; si no existen procesos en espera entonces $s := s + 1$ y el proceso que efectuó SEND(s) continua.

Como se puede observar la variable s contiene en todo instante el número de datos existentes en el acumulador.

Observaciones:

- 1) Si un semáforo, solamente asume los valores 0 y 1, es llamado un semáforo binario; si toma un número arbitrario de valores enteros no negativos, es llamado semáforo general;

- 2) WAIT y SEND son las únicas operaciones permitidas sobre un semáforo después de que dicho semáforo ha sido inicializado con la operación de inicialización. Estas tres operaciones se implantarán en el capítulo III;
- 3) La definición de SEND(s) no especifica cuál proceso es reiniciado, en el caso de que más de un proceso haya sido suspendido por el mismo semáforo, estos procesos en espera, forman la 'cola de procesos en espera'. El proceso que es reiniciado depende del esquema empleado (FIFO, por prioridad, etc.).

Con semáforos sincronizamos el acceso a un recurso compartido por varios procesos, es decir efectuamos exclusión mutua entre los procesos. Es responsabilidad del programador ejecutar las operaciones sobre un semáforo en la secuencia correcta para evitar resultados imprevisibles, por ejemplo si un proceso efectúa un WAIT(s), otro proceso deberá efectuar en el momento oportuno un SEND(s). En el capítulo III se ilustra el uso de semáforos en varios tipos de problemas de concurrencia.

En resumen se puede decir que un semáforo es una herramienta de sincronización entre procesos que comparten un recurso y está compuesto de una variable de

tipo entero y una cola en la cual los procesos esperan por el envío de una señal. La variable toma dos valores si se trata de un semáforo binario o toma varios valores mayores o iguales que cero si se trata de un semáforo general. La cola contiene los procesos que han efectuado una operación WAIT cuando la variable tenía un valor de cero. En nuestra implantación del capítulo III y IV la cola del semáforo se construye a base de punteros.

1.5.3 SECCIONES CRITICAS

Exclusión mutua debe existir solo cuando varios procesos usan un mismo recurso, cuando los procesos realizan operaciones sin interaccionar entre ellos pueden continuar ejecutándose concurrentemente. Cuando un proceso está haciendo uso de un recurso que es compartido por varios procesos, se dice que está en su sección crítica (SC). Una SC debe ejecutarse lo más rápidamente posible y un proceso no debe quedar bloqueado dentro de una SC.

Se usa la notación:

VAR r :SHARED T;

para declarar una variable común o compartida r de tipo T. Procesos concurrentes pueden acceder a secciones

críticas por medio de una estructura llamada región crítica que se declara como sigue:

REGION v DO S;

lo cual significa que mientras un proceso está ejecutando S, ningún proceso puede acceder a v.

Se deben hacer las siguientes asunciones con respecto a secciones críticas sobre una variable v:

- 1) Si un proceso requiere entrar en una SC, lo hará dentro de un tiempo finito.
- 2) Solo un proceso a la vez puede estar dentro de una SC.
- 3) Un proceso permanece dentro de una SC por un tiempo finito.
- 4) Cuando una SC está ocupada por un proceso, otros procesos que requieren entrar a esa SC tienen que esperar, y lo hacen en una 'cola de espera'.

El proceso de la cola de espera que entra a la SC es escogido de acuerdo a diferentes criterios, por ejemplo: esquema de prioridades, arreglo FIFO ,etc.. Con estas nuevas herramientas se puede resolver el problema de exclusión mutua y recursos compartidos presentado anteriormente. El siguiente algoritmo resuelve el

problema para n procesos.

```

VAR R :SHARED BOOLEAN;

COBEGIN
    P1' REPEAT
        REGION R DO usa recurso;
        P1 no esta en uso del recurso
    FOREVER
    *
    *
    Pn' REPEAT
        REGION R DO usa recurso;
        Pn no esta en uso del recurso
    FOREVER
COEND

```

Una región crítica tal como hemos visto en esta sección es una estructura suministrada por el propio lenguaje (en este caso Pascal Concurrente), de modo que el programador no interviene en la sincronización para la entrada y salida a la sección crítica como es el caso cuando se trabaja con semáforos. En este caso es el compilador el que efectúa exclusión mutua.

Una región crítica puede ser también implantada utilizando semáforos. Por cada declaración:

```
VAR v :SHARED T;
```

se declara una variable v-semáforo de tipo SEMAPHORE

initializada a 1 (es decir el primer proceso que efectúa un WAIT(v-semáforo) no es suspendido). Por cada estructura

REGION v DO S;

se genera el siguiente código por medio del compilador correspondiente:

WAIT(v-semáforo);

S;

SEND(v-semáforo);

de lo que se infiere que exclusión mutua es efectuada.

1.5.4 MONITORES

Un semáforo es una herramienta de sincronización que puede ser considerada de un nivel elemental. Un monitor es una herramienta de sincronización más estructurada, es decir la sincronización a una sección crítica y exclusión mutua son efectuadas por el propio lenguaje y Módula-2, como veremos, posee el medio de construir monitores.

Un monitor consiste de procedimientos y un (o varios) tipo(s) de dato(s); así si M es el único monitor que puede acceder a una variable v (por medio de sus procedimientos) entonces está garantizada exclusión mutua del acceso a v, ya que la entrada a M por un proceso excluye a otros procesos; además, ya que las únicas operaciones sobre v están dentro de M, se asegura

que no se hará accidentalmente ninguna asignación o prueba sobre dicha variable.

Otra idea en torno a monitores, es la construcción de datos estructurados o 'abstracción de datos'. Se puede crear una estructura consistente de una declaración de variables y de procedimientos que definen las únicas operaciones legales sobre dichas variables, este monitor puede ser compartido por varios procesos. Módula-2 puede crear datos y monitores bajo ese concepto (capítulo III).

Un monitor está formado por un conjunto de variables y un conjunto de procedimientos que operan sobre dichas variables. Los procedimientos pueden ser usados por varios procesos que requieran operar sobre una variable compartida por ellos, la misma que será la variable declarada en el monitor.

Solamente un proceso a la vez puede entrar al monitor. Otros procesos que requieran entrar esperan fuera de él, y uno de ellos es habilitado para entrar cuando el monitor está libre. Un proceso puede requerir seguir esperando fuera del monitor, hasta que se cumpla cierta condición. Se introduce entonces una variable de tipo semáforo y las operaciones WAIT y SEND se aplicarán

sobre esa variable. Cuando un proceso ejecuta un WAIT, se inserta en la cola y un SEND remueve un proceso de la cola, el mismo que puede entrar al monitor.

Si varios procesos requieren acceder un recurso, que debe ser utilizado por uno a la vez, el siguiente algoritmo, que representa a un monitor bien elemental puede ser usado.

```
MONITOR asigna_recurso;
VAR recurso_en_uso:BOOLEAN;
    recurso_libre:semáforo;

PROCEDURE obtiene_recurso;
BEGIN
    IF recurso_en_uso THEN
        WAIT(recurso_libre);
    recurso_en_uso:=TRUE
END;

PROCEDURE devuelve_recurso;
BEGIN
    recurso_en_uso:=FALSE;
    SEND(recurso_libre)
END;

BEGIN
    recurso_en_uso:=false;
END;
```

este monitor funciona como un semáforo binario.

Consiste de dos procedimientos: `obtiene_recurso` y `devuelve_recurso`, los mismos que pueden ser usados por varios procesos que comparten un recurso. El procedimiento `obtiene_recurso` verifica si la variable `recurso_libre` es falsa, si es así entra a la sección crítica, si es verdadera espera que el recurso sea liberado por otro proceso. El procedimiento `devuelve_recurso` asigna falso a la variable `recurso_libre` y efectúa un SEND para avisar de este evento a un proceso en espera que haya previamente efectuado una operación WAIT.

En el capítulo III se presentan varios monitores construidos en Módula-2 que 'exportan' un tipo de dato y varios procedimientos.

1.6 BLOQUEO DE PROCESOS

Un proceso en un sistema de multiprogramación, se dice que está en estado de bloqueo ('deadlock') si está esperando por un evento que no ocurrirá en un tiempo finito. En la sección 1.4.2 se presenta un caso de bloqueo de procesos, en donde los procesos comparten un recurso.

Se han establecido las siguientes cuatro condiciones para que exista un bloqueo:

- 1) Los procesos exigen el control exclusivo de los recursos.
- 2) Los procesos mantienen recursos ya asignados a ellos mientras esperan por otros recursos.
- 3) Recursos no pueden ser removidos de los procesos que los usan, hasta que esos procesos los requieran.
- 4) Cada proceso de una cadena circular mantiene un recurso requerido por el siguiente proceso.

En el capítulo III se considera la posibilidad de bloqueo de procesos y se sugiere la forma de evitarlo si varios procesos interactúan entre sí.

1.7 ALGORITMOS DE PROGRAMACION CONCURRENTE

En esta parte se introducen tres algoritmos muy tradicionales en programación concurrente, que ilustran problemas de sincronización, exclusión mutua, bloqueo de procesos, etc. En el capítulo III, se presentan las respectivas soluciones en Modula-2.

1.7.1 PROBLEMA PRODUCTOR-CONSUMIDOR

Este problema analiza y resuelve las situaciones que surgen cuando dos procesos interactúan entre sí a través

de una área de memoria común. El proceso llamado productor envía datos a un acumulador y el proceso llamado consumidor recoge dichos datos del acumulador. Como se vió en la sección 1.5.2, los dos procesos tienen que sincronizar el acceso a la memoria común de modo que exista exclusión mutua entre ellos y que se cumplan las reglas de comunicación indicadas. El problema se puede resolver usando cualquiera de las herramientas presentadas en secciones precedentes. En la sección 3.1 está resuelto en Módula-2.

1.7.2 PROBLEMA LECTORES Y ESCRITORES

Este problema ilustra la sincronización entre varios procesos llamados lectores y escritores. Los lectores leen información de una área de memoria común y los escritores escriben información en la misma área, los lectores pueden acceder a la sección crítica al mismo tiempo, es decir no se requiere exclusión mutua entre ellos; entre los escritores sí debe existir exclusión mutua así como entre lectores y escritores. En la sección 3.2 se analiza el problema en Módula-2.

1.7.3 PROBLEMA DE LOS FILOSOFOS

En este problema surgen situaciones de recursos

compartidos, exclusión mutua, etc. y para su solución se utilizan las herramientas analizadas en las secciones precedentes. La descripción es la siguiente: cinco filósofos se sientan a una mesa con cinco platos y cinco tenedores, cada filósofo puede comer solo si están libres los dos tenedores adyacentes a su plato, si no es así espera por ellos. Es decir los filósofos comparten los tenedores y antes de tomar uno verifican si está libre. Luego de usar un tenedor, el filósofo avisa a otros de este evento. Cada filósofo pasa por un ciclo representado como:

```
REPEAT  
    pensar;  
    comer;  
FOREVER;
```

En la sección 3.3 este problema es resuelto en Módula-2.

CAPITULO II INTRODUCCION A MODULA-2

2.1 INTRODUCCION

Este capítulo describe las características que posee Módula-2 para la implantación de programación concurrente. No se toman en consideración tópicos como sintaxis, declaración de tipos, estructuras de control, y otros debido fundamentalmente a que los mismos son muy similares a los que posee el lenguaje Pascal, que es considerado el ancestro de Módula-2. En este capítulo se revisan precisamente aquellas propiedades de Módula-2 que lo diferencian de Pascal, tales como el concepto de módulos, corrutinas, manejo de interrupciones y otros.

2.2 MODULOS

Módulos es una de las características más importantes de Módula-2 y una de las que lo distinguen de Pascal. Hay cuatro clases de módulos: módulos-programa (o simplemente programas), módulos de librería, que constan de dos partes: módulo de definición, módulo de implantación y módulos internos.

Los programas son unidades completas y pueden 'importar' objetos (tipos, variables, constantes, procedimientos).

de módulos de librería. Los módulos de librería son creados para 'exportar' objetos a otros módulos. Los módulos internos, pueden ser declarados dentro de programas y módulos de implantación. La sintaxis de un módulo se indica en la figura 2.1.

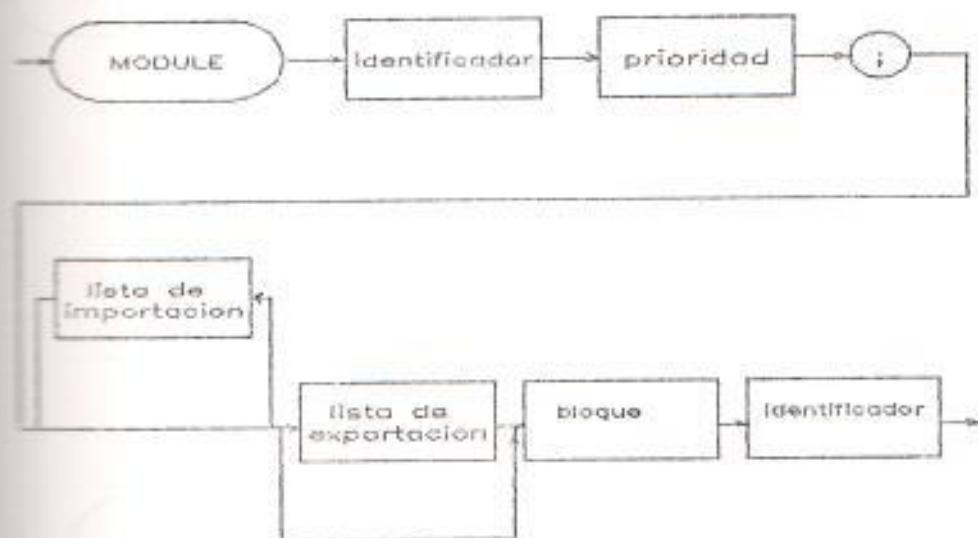
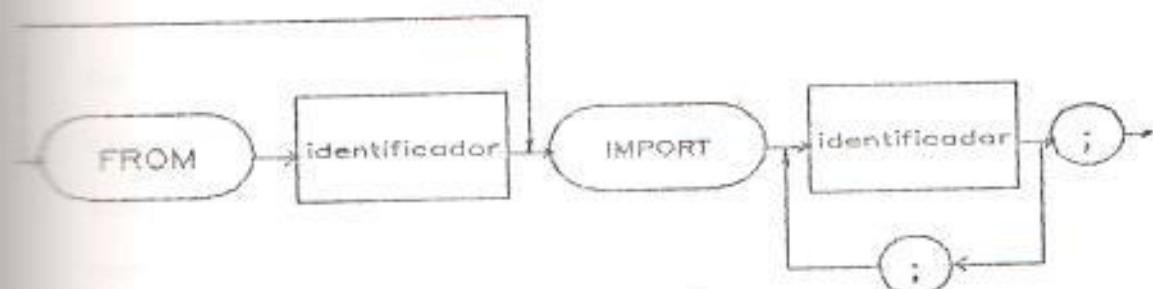


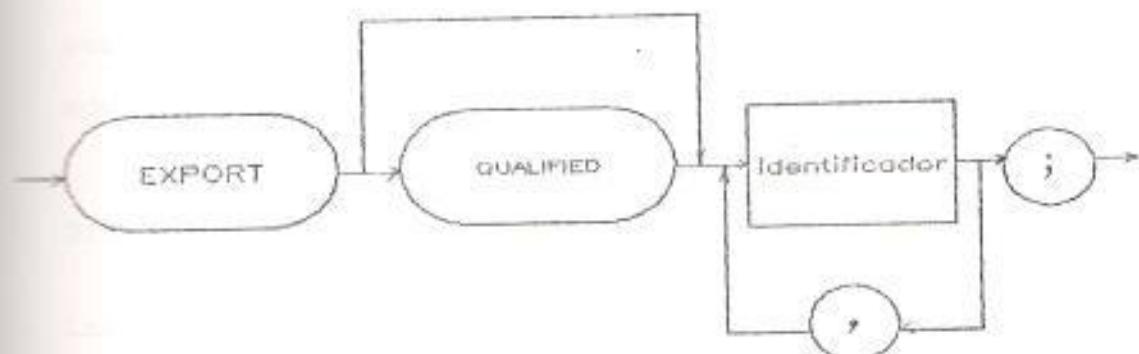
Figura 2.1 Sintaxis de un Módulo

Un identificador es un grupo de caracteres usado para nombrar un objeto que puede ser una constante, una variable, un procedimiento o un módulo. Un bloque este compuesto de declaraciones de constantes, tipos, variables, procedimientos y del conjunto de instrucciones que conforman el programa. Módulos usan 'listas de importación' para especificar objetos usados de otros módulos y 'lista de exportación' para declarar los objetos que pueden ser usados por otros módulos. La

sintaxis de lista de importación y lista de exportación se indica en la figura 2.2.



(a) Lista de Importación



(b) Lista de Exportación

Figura 2.2 Sintaxis de Listas de Importación y Exportación

Programas pueden importar objetos solamente de módulos de librería; algunos módulos de librería son suministrados por el Sistema para operaciones

frecuentemente usadas, como entrada y salida de datos (módulo `inOut`), funciones matemáticas (módulo `MATHLIB`), etc.

Se puede omitir `FROM` y el identificador que le sigue, con lo que la lista quedaría como: `IMPORT identificador(s);` lo que significa que importamos el nombre del módulo y por consiguiente todos sus identificadores. Ejemplo: si un módulo `M` exporta los objetos `a,b,c`, y si un módulo `N` contiene la lista `IMPORT M;` significa que los objetos `a,b,c` deben ser referenciados en `N` como `M.a,M.b,M.c`; esta facilidad permite importar iguales identificadores de diferentes módulos sin dar lugar a ambigüedades de nombres.

2.2.1 DEFINICION E IMPLANTACION

Los módulos de librería constan de módulo de definición y módulo de implantación. El módulo de definición especifica los objetos que son exportados a otros módulos, puede contener también una lista de objetos importados. Este módulo contiene toda la información que un usuario del módulo necesita conocer, indica lo que hace un módulo pero no cómo lo hace. La lista de exportación puede contener constantes, tipos, variables, procedimientos. La figura 2.3 muestra la sintaxis del módulo de definición.

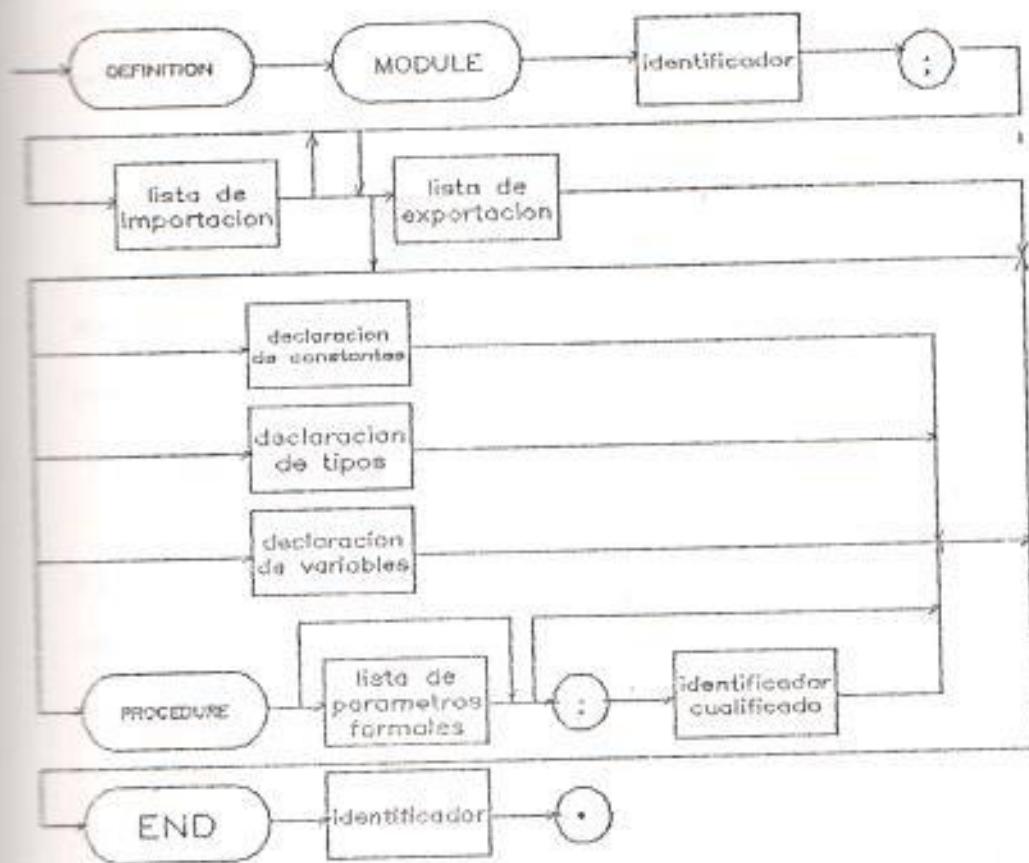


Figura 2.3 Sintaxis de un Módulo de Definición

Los tipos pueden ser declarados como 'opacos' y 'transparentes'. Es opaco cuando se especifica solo su identificador en el módulo de definición, por ejemplo 'TYPE Pila', la estructura de Pila debe ser implantada en el módulo de implantación. Si el tipo Pila está en la lista de exportación, objetos de tipo Pila pueden ser declarados en módulos que importan este tipo. Un tipo

es transparente cuando su estructura completa es especificada en el módulo de definición, si es incluido en la lista de exportación, todos sus componentes también son exportados, por ejemplo si un RECORD es declarado como transparente, todos sus campos son exportados sin que sea necesario incluirlos en la lista. Modula-2 considera a todos los tipos opacos como de tipo puntero, es decir si se declara un tipo opaco, este será necesariamente un puntero a algún objeto.

El módulo de implantación provee la implantación de los objetos declarados en el módulo de definición; tipos opacos, procedimientos, variables y constantes son implantados en este módulo. La sintaxis del módulo de implantación es igual a un módulo de programa, excepto en la palabra IMPLEMENTATION.

Si en el módulo de definición se incluye una lista de importación, ésta debe ser declarada también en el correspondiente módulo de implantación. Un módulo de implantación puede contener constantes, tipos, variables y procedimientos no declarados en el correspondiente módulo de definición, estos objetos son de uso privado de los procedimientos que sí son exportados (2).

2.3 ENTRADA Y SALIDA SECUENCIAL

Módula-2 no posee sentencias o instrucciones para operaciones de entrada y salida de datos (I/O), en cambio provee módulos 'standard' de librería cuyos procedimientos ejecutan estas operaciones: módulo InOut, módulo RealInOut, y además presenta facilidades de bajo nivel que permiten realizar operaciones directas de máquina y que están contenidas en un módulo a nivel de compilador (sección 2.5). Algunos de los procedimientos del módulo InOut son:

Read(ch)	procedimientos de lectura
ReadString(s)	
ReadInt(x)	
ReadCard(x)	
Write(ch)	procedimientos de escritura
WriteString(s)	
WriteLn	
WriteInt(x,n)	
WriteCard(x,n)	
OpenInput(s)	procedimientos para
OpenOutput(s)	redirección de I/O
CloseInput	
CloseOutput	

El módulo InOut define una variable de tipo BOOLEAN: 'Done'; después de una operación de lectura 'Done' es verdadera si la lectura es correcta, caso contrario es falsa. Después de un procedimiento de redirección (OpenInput, OpenOutput) 'Done' es verdadera si un nuevo archivo fue abierto, de otro modo es falsa. Si no se especifica otra cosa, los procedimientos descritos operan sobre los archivos de I/O asumidos por InOut (normalmente teclado para entrada y terminal para salida). Los procedimientos de redirección permiten leer y escribir en otros archivos. El procedimiento:

```
OpenInput(n:ARRAY OF CHAR);
```

cierra el archivo de entrada actual y abre otro archivo cuyo nombre es especificado por el parámetro n, es decir las siguientes entradas son leídas de este archivo. El procedimiento CloseInput cierra el archivo actual de entrada y re-abre el anterior (teclado). El procedimiento:

```
PROCEDURE OpenOutput(n:ARRAY OF CHAR);
```

permite hacer todas las operaciones de escritura en el archivo especificado por el parámetro n. El procedimiento CloseOutput cierra el archivo abierto por OpenOutput.

2.4 PROCESOS EN MODULA-2

Modula-2 posee los medios necesarios para la implantación de programas concurrentes. Lo más importante para esto es la coordinación entre ellos, es decir su interacción sin que un proceso sea afectado por otro. Coprocesamiento es una técnica que ayuda a implantar coordinación entre procesos; dos mecanismos son esenciales para implantar la técnica de coprocesamiento: un medio de identificar y ejecutar un programa y establecerlo como un proceso o corutina, y un método que permite a cada corutina comunicarse con otra. Módula-2 posee las herramientas para elaborar esos mecanismos.

2.4.1 CORRUTINAS

Un procedimiento es un subprograma, es decir que está subordinado a un programa; cuando el procedimiento es llamado, inicia su ejecución y al término de la misma el control regresa al programa principal. Se pueden definir dos procedimientos sin esa relación de jerarquía, es decir ninguno está subordinado al otro, estos procedimientos así definidos son corrutinas, y se ejecutan cooperativamente, cada una es capaz de suspender su ejecución en un instante dado para que otra corutina reinicie (o inicie) su ejecución. Del mismo

modo como un programa llama a un procedimiento explícitamente, cada corriputina debe asignar el procesador a otra corriputina de un modo explícito y cada corriputina reinicia su ejecución en el punto en que fue interrumpida.

2.4.2 EL TIPO PROCESS

En Módula-2 un proceso (o corriputina) es un procedimiento sin parámetros y que no es contenido en otro procedimiento; el proceso puede interrumpir su ejecución por un espacio dado de tiempo.

El tipo de dato PROCESS, que es exportado del módulo SYSTEM (sección 2.5), provee el medio de declarar procesos dentro de un programa. Un proceso tiene asociado con él un procedimiento y un espacio para almacenamiento de sus variables locales, condiciones que lo habilitan para iniciar su ejecución. Una variable de tipo PROCESS contiene el estado de los registros internos del procesador del proceso en ejecución.

Cualquier número de variables de tipo PROCESS pueden ser declarados. Dentro de un módulo que importa el tipo PROCESS, pueden hacerse declaraciones de variables de este tipo de la manera usual, ej.:

```
VAR a,b: PROCESS;
```

Una variable de tipo PROCESS contiene la información de cada proceso que permite iniciar su ejecución, o reiniciarla en caso de que haya sido suspendida.

2.4.3 EL PROCEDIMIENTO NEWPROCESS

El procedimiento NEWPROCESS (que es importado del módulo SYSTEM) crea una corriputina. Se lo define como:

```
PROCEDURE NEWPROCESS (P:PRDC,D:ADDRESS,E:CARDINAL,VAR  
np:PROCESS);
```

el primer parámetro es el procedimiento creado como corriputina, PRDC es un tipo predefinido en Módula-2, D da la dirección en memoria del espacio asignado al proceso, E es el espacio de memoria en BYTES asignado al proceso creado y en el cual almacena sus variables locales, el parámetro np es la variable de tipo PROCESS que tiene la información necesaria para iniciar (o reiniciar cuando haya sido interrumpida) la ejecución de la corriputina P; es decir después de NEWPROCESS, np contiene la descripción inicial del estado de los registros del procesador para el proceso o corriputina P.

2.4.4 EL PROCEDIMIENTO TRANSFER

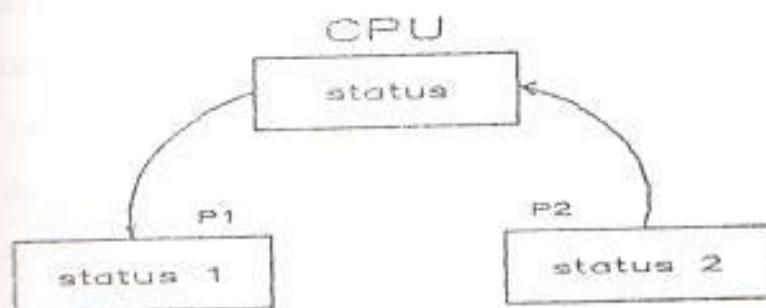
El procedimiento TRANSFER (exportado del módulo SYSTEM) transfiere el control del procesador de un proceso a otro. Es definido como sigue:

```
PROCEDURE TRANSFER (VAR p1:PROCESS, VAR p2:PROCESS);
```

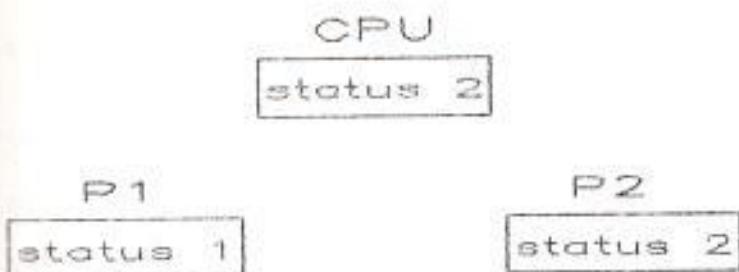
el primer parámetro representa al proceso actual que suspende su ejecución, el segundo parámetro representa al proceso que reinicia (o inicia) su ejecución. Estas variables son originalmente inicializadas con NEWPROCESS.

En el parámetro p1 se guarda la información del proceso actual que suspende su ejecución y, el estado del proceso que reinicia (o inicia) su ejecución se obtiene del parámetro p2. El estado de cada proceso es creado por NEWPROCESS y asignado al parámetro de tipo PROCESS de ese procedimiento.

El procedimiento TRANSFER es efectuado en Módula-2 de modo que primero se accede a la información del segundo parámetro y luego se guarda la información del proceso actual en el primer parámetro. Podemos visualizar la operación de TRANSFER (p1, p2) como:



antes de que el procesador reinicie su operación tenemos:



TRANSFER guarda los registros del procesador correspondientes al proceso actual en la variable p1 y luego carga esos registros con los valores obtenidos de la variable p2.

2.4.5 EL MODULO PROCESSES

El módulo PROCESSES provee una abstracción de sincronización entre procesos. Exporta un tipo de dato y varios procedimientos para sincronizar procesos, permitiendo la asignación del procesador a cada uno de ellos.

Los módulos de definición e implantación son los siguientes.

DEFINITION MODULE Processes;

(* este módulo provee una abstracción de sincronización de procesos *)

FROM SYSTEM IMPORT

(* tipo *) ADDRESS,PROCESS,

(* proc *) TSIZE,NEWPROCESS,TRANSFER;

EXPORT QUALIFIED

(* tipo *) SIGNAL,

```

(* proc *) Init,SEND,WAIT,Awaited, (* para señales *)
           StartProcess;          (* para procesos *)
TYPE
  SIGNAL;
(*
- SIGNAL es el medio de sincronización entre
procesos. Cualquier variable de tipo SIGNAL debe
ser inicializada por medio del procedimiento Init
antes de usarla con cualquier procedimiento de este
módulo.

*)
PROCEDURE Init(VAR s:SIGNAL (*out *));
(*
- Inicializa un objeto de tipo SIGNAL.
*)
PROCEDURE SEND(VAR s:SIGNAL (*in/out*));
(*
- Envía una señal. Un proceso esperando por s,
reanuda su ejecución.
Si no hay procesos en espera, continúa el proceso que
ejecutó SEND.
*)
PROCEDURE WAIT(VAR s: SIGNAL (*in/out*));
(*
- Espera que otro proceso efectue un SEND para
reiniciar la ejecución.
)

```

```

*)

PROCEDURE Awaited( s:SIGNAL (*in *):BOOLEAN;
(*
- Comprueba si hay procesos esperando por una señal. Es TRUE si hay al menos un proceso esperando.

*)

PROCEDURE StartProcess(p:PROC (*in *); n:CARDINAL
(*in *));

(*
- Inicia un nuevo proceso, con espacio de memoria de n BYTES. El proceso creado ejecuta el procedimiento p.

*)

END Processes.

IMPLEMENTATION MODULE Processes;
FROM SYSTEM IMPORT
    (* tipo *) ADDRESS,PROCESS,
    (* proc *).TSIZE,NEWPROCESS,TRANSFER;

FROM Storage IMPORT
    (* proc *) ALLOCATE;

TYPE
    SIGNAL=POINTER TO ProcessDescriptor; (* puntero al descriptor de cada proceso *)
    ProcessDescriptor=RECORD

```

```

next :SIGNAL;      (* lista circular de procesos *)
queue: SIGNAL;     (* cola de procesos en espera *)
cor :PROCESS;      (* registros de cada proceso *)
ready:BOOLEAN;     (* estado del proceso *)

END;

VAR

cp:SIGNAL;      (* esta variable apunta en cualquier
instante al descriptor del proceso actual *)

PROCEDURE StartProcess( P:PROC; n:CARDINAL);
VAR s0 :SIGNAL;
    wsp:ADDRESS;
BEGIN
    s0:=cp;
    ALLOCATE(wsp,n);
    ALLOCATE(cp,TSIZE(ProcessDescriptor));
    WITH cp^ DO
        next:=s0^.next;
        s0^.next:=cp;
        ready:=TRUE;
        queue:=NIL;
    END;
    NEWPROCESS ( P,wsp,n,cp^.cor );
    TRANSFER(s0^.cor, cp^.cor)
END StartProcess;

PROCEDURE SEND(VAR s:SIGNAL);
VAR s0: SIGNAL;

```

```
BEGIN
  IF s<>NIL THEN
    s0:=cp;
    cp:=s;
    WITH cp^.DO
      s:=queue;
      ready:=TRUE;
      queue:=NIL;
    END;
    TRANSFER(s0^.cor,cor,cp^.cor);
  END SEND;

PROCEDURE WAIT(VAR s:SIGNAL);
VAR s0,s1:SIGNAL;
BEGIN
  IF s=NIL THEN s:=cp
  ELSE
    s0:=s;
    s1:=s0^.queue;
    WHILE s1<>NIL DO
      s0:=s1;
      s1:=s0^.queue;
    END;
    s0^.queue:=cp;
  END;
  s0:=cp;
  REPEAT cp:=cp^.next UNTIL cp^.ready;
  IF cp=s0 THEN (* DEADLOCK *) HALT END;
END;
```

```

s0^.ready:=FALSE;
TRANSFER(s0^.cor, cp^.cor)
END WAIT;

PROCEDURE Awaited(s:SIGNAL):BOOLEAN;
BEGIN
  RETURN (s<>NIL)
END Awaited;

PROCEDURE Init(VAR s:SIGNAL);
BEGIN
  s:=NIL;
END Init;

BEGIN (* inicialización de variables *)
  ALLOCATE(cp,TSIZE(ProcessDescriptor));
  WITH cp^ DO
    nextbs:=cp;
    ready:=TRUE;
    queue:=NIL;
  END
END Processes.

```

El módulo PROCESSES permite sincronización entre procesos o corutinas por medio del tipo SIGNAL, que es un puntero al descriptor de cada proceso. El tipo 'ProcessDescriptor' es un bloque de información referente a cada proceso (descriptor de proceso), este bloque es equivalente al PCB indicado en la sección 1.2.1.

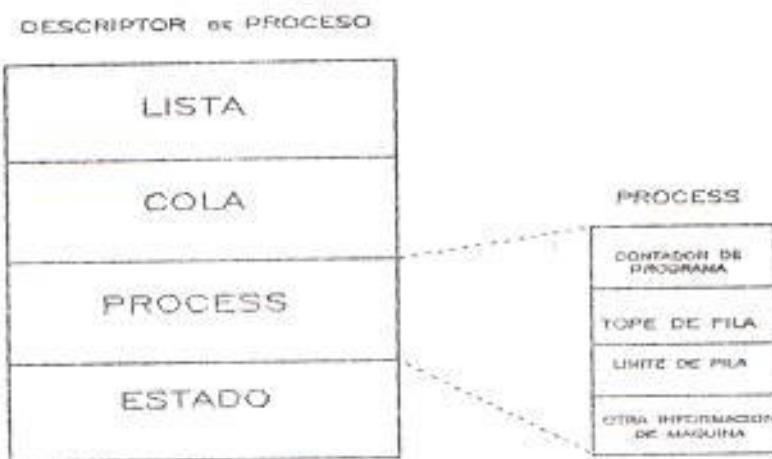


Figura 2.4 Descriptor de Proceso

Es decir los procesos son representados por su correspondiente descriptor de proceso y el módulo PROCESSES opera sobre él (no sobre el proceso).

El descriptor de procesos consta de cuatro partes: 1) el campo 'next' que es de tipo SIGNAL, forma la cola de procesos listos (CPL), esta cola está formada por los descriptores de procesos que están listos para iniciar o reiniciar su ejecución, 2) el campo 'queue' que es de tipo SIGNAL y forma la cola de procesos en espera (CPE) de una señal, 3) el campo 'cor' que es de tipo PROCESS, almacena la información sobre el proceso que es interrumpido para su posterior reinicio y 4) el campo 'ready' que es de tipo BOOLEAN indica si un proceso está

listo para iniciar o reiniciar su ejecución, en cuyo caso tiene un valor de TRUE, e indica también si un proceso está en la cola de procesos en espera (o en estado bloqueado), en cuyo caso tiene el valor FALSE.

Las operaciones sobre señales son dos: SEND y WAIT, además de la operación 'Init' que inicializa la señal (sección 1.5.2).

La operación SEND(s) toma el primer elemento de la cola apuntada por s y transfiere el procesador a ese proceso. Si no hay procesos en espera, el proceso actual identificado por cp continúa su ejecución.

La operación WAIT(s) espera a que un proceso envíe la señal s; inserta el proceso actual al final de la cola apuntada por s, y transfiere el procesador al primer elemento que esté listo para reiniciar su ejecución. Si no existen procesos listos se produce un bloqueo ('deadlock', sección 1.6), lo cual se efectúa con una instrucción HALT. Las operaciones SEND y WAIT actúan de acuerdo a un esquema FIFO, es decir el procesador se asigna al primer proceso de la cola. Se puede trabajar a base de otros criterios por ejemplo: por prioridades, esquema de pila, etc.. SEND reinicia el primer proceso de la CPE, WAIT inserta el proceso al

final de la CPE y asigna el procesador al primer proceso de la CPL.

El procedimiento 'Init' inicializa una señal.

El procedimiento 'Awaited(s)' es verdadero si hay al menos un proceso esperando en la cola s.

El procedimiento 'StartProcess(p,n)' inicia la ejecución de una corriputina P con un espacio de memoria de n BYTES en donde almacena sus datos y variables. La corriputina P es asociada con una variable de tipo PROCESS cuando se ejecuta NEWPROCESS. 'StartProcess' inserta el descriptor de proceso ('ProcessDescriptor') del nuevo proceso o corriputina en la lista de otros procesos creados, para esto utiliza el campo 'next' del descriptor de proceso. Notar que la lista de descriptores formada es circular.

En el siguiente capítulo utilizaremos el módulo PROCESSES para resolver varios problemas de concurrencia de procesos.

2.5 FACILIDADES DE BAJO NIVEL

Módula-2 permite trabajar con localidades de memoria, manipular las direcciones físicas de esas localidades, posee procedimientos para transferencia de datos, por ejemplo de entero a cardinal o de cardinal a entero y otras facilidades de bajo nivel (nivel de máquina).

Módula-2 provee el módulo SYSTEM que exporta tipos y procedimientos para trabajar en nivel de máquina. El módulo de definición de SYSTEM puede ser considerado como:

```
DEFINITION MODULE SYSTEM;
  EXPORT QUALIFIED
    (*tipos*) BYTE,WORD,ADDRESS,PROCESS,
    (*proces*)
    ADR,SIZE,TSIZE,NEWPROCEGS,TRANSFER,IOTRANSFER;
    TYPE BYTE;
    (*- Las variables de tipo BYTE almacenan
     valores de un byte.
    *)
    TYPE WORD;
    (*- Variables de tipo WORD contienen valores
     cuya longitud es de una palabra direccionable
     de memoria.
    *)
    TYPE ADDRESS;
    (*- Variables de tipo ADDRESS contienen
     direcciones de memoria. Son compatibles con
     todas las variables de tipo puntero.
    *)
    TYPE PROCESS;
    (*- Variables de tipo PROCESS se usan para
     designar subrutinas. Almacenan la información
```

```
de la corrutina cuando es suspendida.
```

```
*)
```

```
PROCEDURE ADR(x:cualquier tipo):ADDRESS;
```

```
(-- Retorna la dirección de la variable x.
```

```
*)
```

```
PROCEDURE SIZE(VAR v:cualquier tipo):CARDINAL;
```

```
(-- Retorna el tamaño en bytes de la variable
```

```
v.
```

```
*)
```

```
PROCEDURE TSIZE(t:cualquier tipo):CARDINAL;
```

```
(-- Retorna el tamaño en bytes de la variable
```

```
declarada de tipo t.
```

```
*)
```

```
PROCEDURE NEWPROCESS(P:PROC;D:ADDRESS;N:CARDI-
```

```
NAL;Q:PROCESS);
```

```
(-- Crea una corrutina P .N es el espacio de
```

```
memoria asignado a la corrutina, D es la
```

```
dirección del espacio de memoria N y Q da la
```

```
información para la ejecución de P.
```

```
*)
```

```
PROCEDURE TRANSFER(VAR p,q:PROCESS);
```

```
(-- Suspende la corrutina p e inicia la
```

```
corrutina q.
```

```
*)
```

```
PROCEDURE IOTRANSFER(VAR p,q:PROCESS;Vector:
```

```
ADDRESS);
```

** Suspende la corriente p e inicia la
corriente q. Cuando ocurre la interrupción
dada por el parámetro Vector, el control
regresa a p .

*)

END SYSTEM.

de donde el tipo PROCESS y los procedimientos NEWPROCESS
y TRANSFER se explicaron en la sección 2.4.

Los valores que toma el tipo BYTE son aquellos que ocupan un byte de memoria y los que toma el tipo WORD ocupan una palabra de memoria; solo la operación de asignación es permitida sobre esos tipos. Además si un procedimiento tiene como parámetro uno de tipo ARRAY OF WORD o ARRAY OF BYTE el parámetro actual puede ser de cualquier tipo y longitud.

El tipo ADDRESS puede ser considerado como equivalente a POINTER TO WORD, es decir los valores que puede tomar son direcciones de memoria y es compatible con todos los tipos puntero, con el tipo CARDINAL, y las operaciones que pueden ser hechas sobre variables de este tipo incluyen: dereferencia de punteros, asignación, operaciones aritméticas y relaciones.

El procedimiento ADR es un procedimiento de función que

entrega la dirección de su parámetro, que debe ser una variable.

El procedimiento SIZE es un procedimiento de función que entrega el tamaño de su parámetro el mismo que es una variable. El procedimiento TSIZE toma un identificador de tipo como parámetro y da el tamaño de objetos de ese tipo. Estos dos procedimientos dan el tamaño en unidades de memoria, que pueden ser BYTE o WORD (depende de la máquina).

Con estas características, Módula-2 puede manejar memoria ya que puede hacer operaciones aritméticas sobre direcciones; también se puede manipular punteros con operaciones aritméticas. Ej.: si tenemos

VAR dir:ADDRESS;

la sentencia:

dir:=ADR(v)+SIZE(v);

asigna a la variable dir la dirección de memoria inmediatamente siguiente a la variable v.

2.6 MANEJO DE DISPOSITIVOS E INTERRUPCIONES

En esta sección se considera el modo en que Módula-2 interactúa con dispositivos periféricos de entrada/salida. Cada dispositivo trabaja en mapeo de

memoria, es decir dada uno de ellos tiene una dirección determinada. Para asignar una dirección absoluta a un dispositivo en un programa, esas direcciones son referenciadas como variables normales, pero la declaración de esas variables debe proveer la dirección absoluta a la cual esas variables son asignadas por el compilador. La sintaxis de esa declaración está en la figura 2.5.

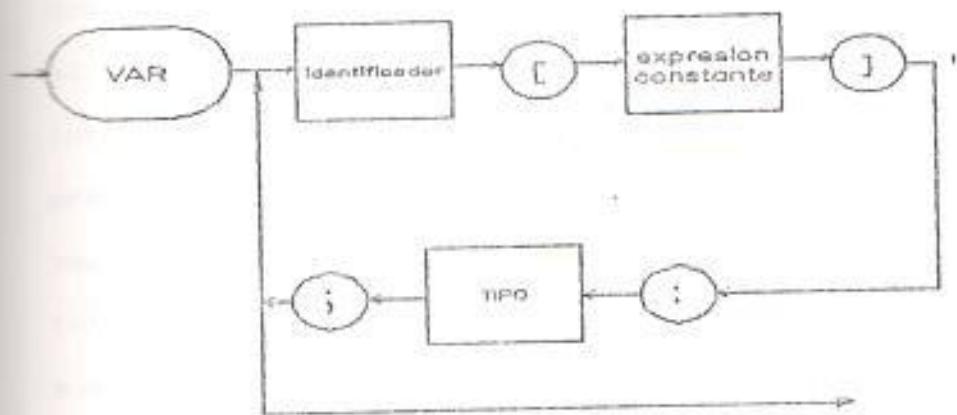


Figura 2.5 Sintaxis de una especificación de memoria absoluta

Modula-2 tiene también un tipo especial que es BITSET, que permite manipular bits dentro de una palabra.

Si un dispositivo envía una interrupción al procesador, este debe parar el proceso que está ejecutando y atender la rutina de interrupción, accediendo a ella por medio del 'vector de interrupción' que tiene una dirección

conocida. Cuando la rutina de interrupción termina, regresa el control al proceso que fue interrumpido; este proceso no tiene forma de conocer cuando ocurrirá una interrupción, ya que es una señal externa la que inicia la transferencia. Módulo-2 provee un modo de declarar una rutina de servicio de interrupción como proceso o corrutina, para esto provee el procedimiento IOTRANSFER:

```
PROCEDURE      IOTRANSFER(VAR      source:PROCESS,VAR  
destination:PROCESS, vía:ADDRESS);
```

donde 'source' es una variable tipo PROCESS que contiene el estado actual de la rutina de interrupción, 'destination' contiene el estado del proceso que fue interrumpido, 'vía' es la dirección del vector de interrupción. La ocurrencia de la interrupción equivale a TRANSFER(destination,source), ejecutado en el proceso que es interrumpido y el control del procesador lo toma la rutina de servicio de la interrupción en el punto inmediatamente después del IOTRANFER.

En el capítulo IV usaremos IOTRANSFER para atender la interrupción del temporizador interno del IBM PC.

CAPITULO III IMPLANTACION DE ALGORITMOS USANDO MODULA-2

Este capítulo aborda tres problemas tradicionales en programación concurrente, los cuales son implantados usando Modula-2. Para la solución de estos problemas usaremos el módulo PROCESSES, el cual provee un medio de sincronización entre procesos y además soporta quasi-concurrencia. Como veremos en el presente capítulo, el módulo PROCESSES puede ser modificado de acuerdo al problema particular en que será usado; de esta forma podemos considerar el módulo PROCESSES de la sección 2.4.5 como un módulo básico, a partir del cual, el programador realiza las modificaciones o adecuaciones que considere necesario.

3.1 PROBLEMA PRODUCTOR-CONSUMIDOR

Como vimos en el capítulo I, un proceso denominado Productor produce datos y los envía a un acumulador y otro proceso llamado Consumidor, consume dichos datos del acumulador. En esta parte resolveremos los problemas de sincronización que surgen de esta situación. Usaremos un módulo PROCESSES modificado, el cual se presenta a continuación.

```
DEFINITION MODULE Processes;
```

```
EXPORT QUALIFIED
```

```

(* tipo *) SIGNAL,
(* proc *) SEND, WAIT, Awaited, Init,      (* para
señales *)
(* proc *) StartProcess, WaitMain, Terminate; (*
para procesos *)

TYPE SIGNAL;

(*
- SIGNAL es el medio de sincronizar procesos. El
tipo SIGNAL es un puntero a un semáforo
general; antes de usar variables de tipo
SIGNAL, éstas deben ser inicializadas con el
procedimiento 'Init'.
*)

VAR idle: SIGNAL;

(*
- La variable 'idle' es usada por el programa
principal, el cual efectúa WAIT(idle). Cuando
todos los procesos han terminado, el programa
principal es removido de la cola de la señal
'idle'.
*)

PROCEDURE StartProcess(P:PROC;n:CARDINAL);
(*
- 'StartProcess' inicia la ejecución de un proceso; P
es el procedimiento que se ejecuta como proceso y n
es el espacio de memoria en BYTES asignado a ese
procedimiento.
*)

```

```
PROCEDURE SEND(VAR s:SIGNAL);
```

```
(*
```

- Este procedimiento envía una señal a un proceso en espera de dicha señal, el cual reinicia su ejecución. Si no existen procesos en espera de la señal, el procesador es asignado a un proceso de la cola de procesos listos. Si no hay ningún proceso listo en esta cola, el proceso que llamó a SEND continúa su ejecución.

```
*)
```

```
PROCEDURE WAIT(VAR s:SIGNAL);
```

```
(*
```

- El procedimiento WAIT(s) inserta el proceso actual en la cola de la señal s. Luego el procesador es asignado al primer proceso de la cola de procesos listos.

```
*)
```

```
PROCEDURE Awaited(s:SIGNAL):BOOLEAN;
```

```
(*
```

- Este procedimiento verifica si hay procesos en espera de la señal s. Es verdadero si existe al menos un proceso en espera.

```
*)
```

```
PROCEDURE Init(VAR s:SIGNAL);
```

```
(*
```

- Este procedimiento inicializa variables de tipo SIGNAL.

```
    *)
PROCEDURE Terminate;
(*
- Este procedimiento termina un proceso y lo remueve
de la cola de procesos. Si todos los procesos han
terminado, remueve el programa principal de la cola
de la señal 'idle'.
*)
PROCEDURE WaitMain;
(*
- Este procedimiento es llamado por el programa
principal después de que todos los procesos hayan
iniciado su ejecución. Efectúa un WAIT(idle).
*)
END Processes.

IMPLEMENTATION MODULE Processes;
FROM SYSTEM IMPORT
    (* type *) ADDRESS,PROCESS,
    (* proc *) NEWPROCESS,TRANSFER,TSIZE;
FROM Storage IMPORT
    (* proc *) ALLOCATE,DEALLOCATE;
TYPE SIGNAL=POINTER TO semaphore;
    (* señal para sincronizar procesos *)
list=POINTER TO ProcessDescriptor;
    (* variable para colas de procesos *)
semaphore=RECORD
```

```
        value:CARDINAL; (* valor del semáforo *)
        queueesem:list    (* cola de procesos del semáforo *)
    END;

ProcessDescriptor=RECORD
    next:list;      (* cola de procesos listos *)
    queue:list;     (* cola de procesos en espera *)
    cor:PROCESS;   (* registros del proceso *)
    ready:BOOLEAN (* estado del proceso *)
END;

VAR
    cp:list; (* cp apunta en cualquier instante al
proceso en ejecución *)
    numberprocesses:CARDINAL; (* 'numberprocesses'
contiene en cualquier instante el número de
procesos activos *)

PROCEDURE StartProcess(P:PROC;n:CARDINAL);
VAR s0:list;
    wsp:ADDRESS;
BEGIN
    s0:=cp;
    INC(numberprocesses);
    ALLOCATE(wsp,n);
    ALLOCATE(cp,TSIZE(ProcessDescriptor));
    WITH cp^ DO
        next:=s0^.next;
        s0^.next:=cp;
```

```

ready:=TRUE;
queues:=NIL;
END;
NEWPROCESS(P,wsp,n,cp^.cor);
TRANSFER(s0^.cor, cp^.cor)
END StartProcess;

PROCEDURE SEND(VAR s: SIGNAL);
VAR so: list;
BEGIN
IF s^.queuesem <> NIL THEN
  so:=cp;
  cp:=s^.queuesem;
  WITH cp^ DO
    s^.queuesem:=queues;
    ready:=TRUE;
    queues:=NIL;
  END;
  TRANSFER(so^.cor, cp^.cor);
ELSE INC(s^.value);
  so:=cp;
  REPEAT cp:=cp^.next UNTIL cp^.ready;
  IF cp>so THEN TRANSFER(so^.cor, cp^.cor)END;
END;
END SEND;

PROCEDURE WAIT(VAR s: SIGNAL);

```

```

VAR s0,s1: list;
BEGIN
  IF s^.value>0 THEN DEC(s^.value)
  ELSE
    IF s^.queuesem=NIL THEN s^.queuesem:=cp
    ELSE
      s0:=s^.queuesem;
      s1:=s0^.queue;
      WHILE s1<>NIL DO
        s0:=s1;
        s1:=s0^.queue;
      END;
      s0^.queuesem=cp;
    END;
    s0:=cp;
  REPEAT cp:=cp^.next UNTIL cp^.ready;
  IF cp=s0 THEN (* deadlock *) HALT END;
  s0^.ready:=FALSE;
  TRANSFER(so^.cor,cp^.cor);
END;
END WAIT;

```

```

PROCEDURE Awaited(s: SIGNALY: BOOLEAN;
BEGIN
  RETURN (s^.queuesem<>NIL)
END Awaited;

```

```
PROCEDURE Init(VAR s: SIGNAL);
BEGIN
    ALLOCATE(s, TSIZE(semaphore));
    s^.value:=0;
    s^.queuesem:=NIL;
END Init;

PROCEDURE Terminate;
VAR s0,s1:list;
    processout:PROCESS;
BEGIN
    DEC(numberprocesses);
    s0:=cp;
    IF (np=0) AND (Awaited(idle)) THEN
        cp:=cp^.next;
        cp^.ready:=TRUE;
        cp:=s0
    END;
    REPEAT cp:=cp^.next UNTIL cp^.ready;
    IF cp=s0 THEN (* deadlock *) HALT END;
    s1:=s0;
    REPEAT s1:=s1^.next UNTIL s1^.next=s0;
    s1^.next:=s0^.next;
    DEALLOCATE(s0, TSIZE(ProcessDescriptor));
    TRANSFER(processout, cp^.cor)
END Terminate;
```

```

PROCEDURE WaitMain;
BEGIN
  IF numberprocesses>0 THEN WAIT(idle) END;
END WaitMain;
BEGIN (* inicialización del módulo *)
  ALLOCATE(cp,TSIZE(ProcessDescriptor));
  WITH cp^ DO
    next:=cp;
    ready:=TRUE;
    queue:=NIL;
  END;
  numberprocesses:=0;
  init(idle)
END Processes.

```

El módulo PROCESSES permite sincronización entre procesos o corrutinas por medio del tipo SIGNAL, el cual es un puntero a un semáforo. El semáforo está formado por una variable que toma valores mayores o iguales que cero y una cola para procesos, es decir se trata de un semáforo general. La variable se designa como 'value' y la cola se forma manipulando el puntero denominado 'queueasem', el cual apunta a un descriptor de proceso.

La variable 'numberprocesses' contiene el número de procesos activos en cualquier momento. Es incrementada por 'StartProcess' y decrementada por 'Terminate'. Se

entiende por proceso activo en este caso, un proceso que no ha efectuado el procedimiento 'Terminate'.

La variable 'idle' de tipo SIGNAL, es usada por el programa principal, el cual efectúa WAIT(idle), lo que causa que se inserte en la cola de esta señal, de donde es removido cuando todos los procesos han terminado ('numberprocesses' igual a cero). El programa principal es considerado por Modula-2 como una corriputina. En nuestro caso su única función es iniciar los procesos (mediante el procedimiento 'StartProcess') y luego espera que todos ellos terminen. Esto permite que cada proceso termine su ejecución por sí mismo (mediante el procedimiento 'Terminate') sin que ello afecte la ejecución de las otras corriputinas. Si no se tratara el problema de este modo, el programa principal tendría que ser una de las corriputinas y su terminación implicaría necesariamente la terminación de todas las corriputinas restantes.

Las operaciones sobre el semáforo son WAIT y SEND, además de la operación 'Init' la cual inicializa el semáforo (sección 1.5.2). Con estas operaciones se efectúa exclusión mutua entre procesos.

Cuando una operación SEND(s) es efectuada debe haber una transferencia del procesador a un proceso en espera de

esa señal, es decir un proceso que ha efectuado un WAIT(s) previamente. Si hay varios procesos en espera, el proceso que es removido es el primero de la cola (esquema 'fifo': primero que entra primero que sale); si no existen procesos en espera, SEND incrementa la variable 'value' del semáforo y el procesador es asignado al siguiente proceso que está listo en la cola el cual se determina mediante el campo 'ready' ('cp^.ready=TRUE'), si no hay procesos listos, el proceso actual identificado por cp continúa.

La operación WAIT(s) pregunta por el valor de la variable 'value', si es mayor que cero (ha habido por lo menos un SEND(s)) el proceso actual identificado por cp continúa y entra a su sección crítica. Si 'value' es cero, el proceso actual es insertado al final de la cola de s y el procesador es asignado al primer proceso listo para la ejecución, si no existe ningún proceso listo se produce un bloqueo y se efectúa un HALT, lo cual para el programa entero.

Como el semáforo utilizado es un semáforo general, un proceso puede efectuar varios SEND sin que otro proceso haya efectuado un WAIT. El número de SEND efectuados es almacenado en la variable 'value' del semáforo. Así, si un proceso posteriormente ejecuta un WAIT (sobre la misma señal) verifica que 'value' es mayor que cero, la decremente en uno y entra a su sección crítica.

El procedimiento 'Init' inicializa el semáforo, con 'value' igual a cero y 'queuesem' igual a NIL. En ciertas aplicaciones puede ser conveniente inicializar 'value' con un valor de uno.

El procedimiento 'Awaited(s)' es verdadero si hay al menos un proceso en espera, en la cola de s.

El procedimiento 'StartProcess(P,n)' inicia la ejecución de una corutina P con espacio de memoria n BYTES en donde almacena sus datos y estados de sus variables locales. Además incrementa 'numberprocesses'. La corutina P es asociada a una variable de tipo PROCESS cuando se ejecuta NEWPROCESS. 'StartProcess' inserta el descriptor del nuevo proceso en la lista de otros procesos creados, a continuación del descriptor de proceso del proceso que invoca 'StartProcess', que en este caso será el programa principal, para lo cual utiliza el campo 'next' del respectivo descriptor de proceso.

El procedimiento 'WaitMain' es llamado por el programa principal, el cual efectúa WAIT(idle) si 'numberprocesses' es mayor que cero.

El procedimiento 'Terminate' termina un proceso. Lo remueve del uso del procesador. Además decremente 'numberprocesses' y, si todos los procesos han terminado ('numberprocesses'=0) y si el programa principal ha

efectuado WAIT(idle) ('Awaited(idle)=TRUE'), 'Terminate' remueve el programa principal de la cola de la señal 'idle'.

Cuando un proceso llama a 'Terminate', el descriptor de este proceso es excluido de la lista de procesos, para esto utilizamos el procedimiento DEALLOCATE del módulo 'Storage'.

El procedimiento 'Terminate' es llamado únicamente por procesos, cuando estos requieren autoeliminarse. El programa principal no requiere de este procedimiento.

Ahora consideraremos la implantación del acumulador B ('buffer'). B es una sección crítica para cada proceso, y el acceso es sincronizado con semáforos. Aquí usamos un B circular declarado como un arreglo de caracteres. B que es la variable compartida entre los procesos y actúa como medio de comunicación entre ellos, es encapsulado en un módulo junto con los procedimientos que operan sobre B.

La implantación de B no necesita ser conocida por los procesos que lo usan. Para ocultar esta implantación, organizamos B como un módulo de definición e implantación. Una variable compartida que es encapsulada de esta manera es llamada un Monitor. En modula-2 un monitor es un módulo-programa, un módulo

interno o un módulo de implantación con un dígito entre 0 y 7 encerrado entre corchetes que va después del nombre del módulo. Este dígito indica la prioridad del monitor. Prioridades son usadas para controlar la ocurrencia de interrupciones. Es decir, si un proceso está ejecutando un procedimiento importado de un monitor definido con una cierta prioridad, no puede ser interrumpido, salvo el caso en que la interrupción fuera atendida por un monitor definido con una prioridad más alta. A continuación se presenta el monitor para el manejo de B.

```

DEFINITION MODULE BufferCircular;
(* Monitor para manejo del BUFFER. Exporta los
procedimientos depositar y buscar *)
EXPORT QUALIFIED
    (* proc *) depositar, buscar;
PROCEDURE depositar(x:CHAR);
(*
- Deposita un carácter en el 'buffer'.
*)
PROCEDURE buscar(VAR x:CHAR);
(*
- Libera un carácter del 'buffer'.
*)
END BufferCircular.

```

```

IMPLEMENTATION MODULE BufferCircular;

FROM Processes IMPORT
    (* tipo *) SIGNAL,
    (* proc *) SEND,WAIT,Init;

CONST tamaño_buffer=10; (* tamaño del 'buffer' *)
TYPE rango_buffer=0..tamaño_buffer-1;

VAR
    Buffer: ARRAY rango_buffer OF CHAR; (* 'buffer' *
                                         variable compartida *)
    carácter_presente:SIGNAL;          (* señal para
                                         sincronización *)
    espacio_presente: SIGNAL;          (* señal para
                                         sincronización *)
    entrada:rango_buffer;             (* indice para entrada al
                                         'buffer' *)
    salida: rango_buffer;              (* indice para salida del
                                         'buffer' *)
    contadores: CARDINAL;            (* contador para inicializar
                                         señal *)
                                         */

PROCEDURE depositar(carácter:CHAR);
BEGIN
    WAIT(espacio_presente);
    Buffer[entrada]:=carácter;
    entradas:=(entrada+1)MOD tamaño_buffer;
    SEND(carácter_presente);
END depositar;

```

```
PROCEDURE buscar (VAR caracter:CHAR);  
BEGIN  
    WAIT(caracter_presente);  
    caracter:=Buffer[salida];  
    salida:=(salida+1)MOD tamano_buffer;  
    SEND(espacio_presente)  
END buscar;  
  
BEGIN (* inicialización de variables *)  
    entrada:=0;  
    salida:=0;  
    Init(caracter_presente);  
    Init(espacio_presente);  
    FOR contador:=1 TO tamano_buffer DO  
        SEND(espacio_presente);  
    END;  
END BufferCircular.
```

Los procesos no necesitan conocer cómo insertar datos en B y cómo remover datos de él; estas operaciones se realizan por medio de las operaciones depositar y buscar que son exportadas del monitor. En este monitor el acceso a B crea una sección crítica, y la sincronización entre los procesos está oculta en las operaciones depositar y buscar, las mismas que llaman los procedimientos SEND y WAIT apropiados para dicha sincronización. El procedimiento depositar, inserta un carácter en B en la posición dada por la variable

entrada; el procedimiento buscar remueve un carácter de la posición dada por la variable salida.

Para la sincronización entre los procesos se requieren dos variables de tipo SIGNAL (importado del módulo PROCESSES), la una para indicar que un carácter ha sido insertado en B y la otra para indicar que un carácter ha sido removido de B, lo cual es equivalente a decir que un espacio vacío ha sido insertado en B. Estas dos señales son carácter_presente y espacio_presente respectivamente, las cuales son inicializadas con el procedimiento 'Init' (importado del módulo PROCESSES).

El hecho de que las señales incorporen contadores, significa que el número de caracteres y de espacios en B está almacenado dentro de las propias señales (en la variable 'value') en cualquier instante. El módulo de inicialización ejecuta SEND(espacio_presente) un número de veces igual al tamaño de B, en este caso diez, lo que significa que un proceso depositaría diez caracteres en B antes de ser suspendido por la operación WAIT(espacio_presente).

Este B puede ser usado por varios productores y consumidores y en diverso orden.

El productor es considerado como :

```
PROCEDURE productor;
```

```

BEGIN
LOOP
  'produce'
  'deposita'
  .
  .
  .
END
END productor;

```

El consumidor es considerado como :

```

PROCEDURE consumidor;
BEGIN
LOOP
  'busca datos'
  'procesa datos'
  .
  .
  .
END
END consumidor;

```

El siguiente programa ilustra el uso de BufferCircular, en el cual el productor llama a depositar y el consumidor llama a buscar.

```
MODULE BufferCircularDemostración;
```

```
(* Módulo que ilustra el funcionamiento de procesos concurrentes. Se descompone en dos procesos concurrentes: productor y consumidor; cada uno de ellos hace uso del procesador por un determinado período de tiempo *)  
FROM Processes IMPORT  
    (* proc *) StartProcess,WaitMain,Terminate;  
FROM BufferCircular IMPORT  
    (* proc *) depositar,buscar;  
FROM InOut IMPORT  
    (* const *) EOL,  
    (* proc *) WriteLn,WriteString,Write,Read;  
PROCEDURE productor;  
VAR caracter:CHAR;  
BEGIN  
    WriteString(' productor ha empezado');  
    WriteLn;  
LOOP  
    Read(caracter);  
    depositar(caracter);  
    IF caracter=EOL THEN Terminate END;  
END;  
END productor;  
PROCEDURE consumidor;  
VAR caracter:CHAR;  
BEGIN
```

```
WriteString(' consumidor ha comenzado');

WriteLn;

LOOP

buscar(caracter);

Write(caracter);

IF caracter=EOL THEN Terminate END;

END;

END consumidor;

BEGIN (* programa principal *)

WriteString('inicio de BufferCircularDemostración');

WriteLn;

StartProcess(producer,500);

StartProcess(consumidor,500);

WaitMain;

WriteLn;

WriteString('fin de BufferCircularDemostración');

WriteLn;

END BufferCircularDemostración.
```

Nota: no hay una forma precisa de determinar el tamaño de memoria que requiere cada proceso. En este caso un tamaño de 500 BYTES parece razonable.

Antes de analizar este programa recordar que:

- 1) la ejecución de 'StartProcess(p,n)' inserta el proceso actual (PA) en la cola de procesos listos

- (CPL). El procesador se asigna a la corriutina p, e inicia su ejecución.
- 2) un SEND(s) asigna el procesador a un proceso de la cola de s, si no hay procesos en esta cola, el procesador se asigna a un proceso de la CPL y 'value' es incrementada en 1. El PA es insertado en la CPL. Si no hay procesos en las dos colas el PA continúa.
 - 3) un WAIT(s) inserta el PA en la cola de s, y asigna el procesador a un proceso de la CPL, en el caso de que 'value' es cero; si 'value' es mayor que cero el PA decrementa 'value' y continúa.

El programa BufferCircularDemostración es como sigue: el programa principal inicia su ejecución e imprime el mensaje 'inicio de BufferCircularDemostración', luego ejecuta 'StartProcess(productor,500)' e inicia el proceso productor (el programa principal se queda en la CPL) el cual imprime el mensaje 'productor ha empezado', lee un carácter del teclado, digamos 'a', y llama a depositar, aquí se efectúa una operación WAIT(espacio_presente), la cual encuentra que la variable 'value' es mayor que cero y entra a B, deposita el carácter en la posición dada por la variable entrada, que es cero y ejecuta SEND(caracter_presente), esta operación pasa el control al programa principal el

cual ejecuta 'StartProcess(consumidor, 500)', el consumidor se inicia e imprime el mensaje 'consumidor ha comenzado', llama a buscar, ejecuta WAIT(caracter_presente), como previamente hubo un SEND(caracter_presente), entra a B y libera el primer carácter, ejecuta SEND (espacio_presente), con esta operación el control es pasado al productor, el cual lee otro carácter, digamos 'b', llama a depositar, hace un WAIT (espacio_presente), entra al 'buffer', inserta 'b' en la posición dada por entrada que es uno, efectúa SEND (caracter_presente), el control lo toma el programa principal, el cual efectúa 'WaitMain', aquí se ejecuta un WAIT(idle) y esta operación inserta el programa principal en la cola de la señal 'idle' y pasa el control al consumidor el cual imprime el carácter 'a', llama a buscar, toma el segundo carácter que es 'b', etc.

Cada proceso termina cuando encuentra el carácter EOL (importado de InOut) luego de lo cual el programa principal es removido de la cola de la señal 'idle' e imprime el mensaje ' fin de BufferCircularDemostración'.

Como ejemplo escribimos la palabra ESPOL. Los resultados se muestran a continuación, en donde las letras entre comillas aparecen por pantalla, y las letras sin comillas son entradas por teclado.

'inicio de BufferCircularDemostración'

'productor ha empezado'

E

'consumidor ha empezado'

S

'E'

P

'S'

D

'P'

L

'D'

EOL

'L'

'fin de BufferCircularDemostración'

Este programa ilustra cuasi-concurrencia, ya que el procesador es asignado a cada proceso (compartido en el tiempo) y se efectúa la comunicación entre procesos mediante secciones críticas. El tiempo asignado a cada proceso depende de la duración de su sección crítica (en este caso depositar y buscar), y también de las operaciones que ejecuta fuera de ella; por supuesto en este caso el productor retiene mucho tiempo el procesador con la instrucción 'Read(ch)'.

La transferencia del procesador de un proceso a otro

depende solo de la operaciones SEND y WAIT de PROCESSES, las mismas pueden tener otros esquemas. El esquema en este caso es, como se dijo, el esquema 'fifo'. Si usáramos un SEND similar al del módulo PROCESSES de la sección 2.4.5, el productor primero llenará B, efectuará un WAIT (espacio_presente) y el procesador se asignará (recién en este momento) al consumidor.

3.2 PROBLEMA LECTORES Y ESCRITORES

El problema es el siguiente: dos tipos de procesos concurrentes llamados lectores y escritores (L y E) comparten un recurso. Los L pueden usar el recurso simultáneamente (en caso de concurrencia real), pero cada E debe tener acceso exclusivo a dicho recurso. Para este problema utilizaremos un módulo PROCESSES sin contador de señales, es decir un módulo PROCESSES como el de la sección 2.4.5 añadido los procedimientos 'Terminate' y 'WaitMain'. El módulo de definición se puede considerar igual al presentado en la sección 3.1 y a continuación se presenta el correspondiente módulo de implantación.

```
IMPLEMENTATION MODULE Processes;
FROM SYSTEM IMPORT
    (* tipos *) ADDRESS,PROCESS,
```

```
(* proc *) NEWPROCESS,TRANSFER,TSIZE;
FROM Storage IMPORT
    (* proc *) ALLOCATE,DEALLOCATE;
TYPE SIGNAL=POINTER TO ProcessDescriptor;
ProcessDescriptor=RECORD
    next:SIGNAL; (* lista de procesos *)
    queue:SIGNAL; (* cola de procesos en espera *)
    cor:PROCESS; (* registros del proceso *)
    ready:BOOLEAN (* estado del proceso *)
END;
VAR cp:SIGNAL; (* apuntador del proceso actual *)
numberprocesses:CARDINAL; (* número de procesos en la
lista *)
PROCEDURE StartProcess(P:PROC;n:CARDINAL);
VAR s0:SIGNAL;
    wsp:ADDRESS;
BEGIN
    INC(numberprocesses);
    s0:=cp;
    ALLOCATE(wsp,n);
    ALLOCATE(cp,TSIZE(ProcessDescriptor));
    WITH cp^ DO
        next:=s0^.next;
        s0^.next:=cp;
        ready:=TRUE;
        queue:=NIL;
```

```

END;

NEWPROCESS(P,wsp,n,cp^.cor);
TRANSFER(s0^.cor, cp^.cor)
END StartProcess;

PROCEDURE SEND(VAR s:SIGNAL);
VAR s0:SIGNAL;
BEGIN
  IF s<>NIL THEN
    s0:=cp;
    cp:=s;
    WITH cp^ DO
      s1:=queue;
      ready:=TRUE;
      queues=NIL;
    END;
    TRANSFER(s0^.cor, cp^.cor);
  ELSE
    s0:=cp;
    REPEAT cp:=cp^.next UNTIL cp^.ready;
    IF cp>s0 THEN TRANSFER(s0^.cor, cp^.cor)END;
  END;
END SEND;

PROCEDURE WAIT(VAR s:SIGNAL);
VAR s0,s1:SIGNAL;
BEGIN

```

```
IF s=NIL THEN s:=cp
ELSE
  s0:=s;
  s1:=s0^.queue;
  WHILE s1<>NIL DO
    s0:=s1;
    s1:=s0^.queue;
  END;
  s0^.queue:=cp;
END;
s0:=cp;
REPEAT cp:=cp^.next UNTIL cp^.ready;
IF cp=s0 THEN (* 'deadlock' *) HALT END;
s0^.ready:=FALSE;
TRANSFER(s0^.cor, cp^.cor);
END WAIT;

PROCEDURE Awaited( s:SIGNAL):BOOLEAN;
BEGIN
  RETURN (s<>NIL)
END Awaited;

PROCEDURE Init (VAR s:SIGNAL);
BEGIN
  s:=NIL
END Init;
```

```
PROCEDURE Terminate;
VAR s0,s1: SIGNAL;
    processout: PROCESS;
BEGIN
    DEC(numberprocesses);
    s0:=cp;
    IF (numberprocesses=0) AND (Awaited(idle)) THEN
        cp:=cp^.next;
        cp^.ready:=TRUE;
        cp:=s0;
    END;
    REPEAT cp:=cp^.next UNTIL cp^.ready;
    IF cp=s0 THEN (* deadlock *) HALT END;
    s1:=s0;
    REPEAT s1:=s1^.next UNTIL s1^.next=s0;
    s1^.next:=s0^.next;
    DEALLOCATE(s0,TSIZE(ProcessDescriptor));
    TRANSFER(processout,cp^.cor);
END Terminate;

PROCEDURE WaitMain;
BEGIN
    IF numberprocesses>0 THEN WAIT(idle)
END WaitMain;

BEGIN (* inicialización de variables *)
    ALLOCATE(cp,TSIZE(ProcessDescriptor));

```

```

WITH cp^ DO
  next:=cp;
  ready:=TRUE;
  queue:=NIL;
END;
numberprocesses:=0; Init(idle)
END Processes.

```

Cada proceso (lector y escritor) debe pasar por la siguiente secuencia:

- 'solicitar recurso'
- 'usar recurso'
- 'liberar recurso'

Además, se dice que un proceso está 'activo' desde el momento que solicita el recurso hasta que lo libera, y se dice que está en 'ejecución' desde el momento en que recibe el recurso hasta que lo libera; para lo cual se establecen las siguientes variables.

le : número de L en ejecución
 ea : número de E activos

Para la solución del problema se imponen las siguientes condiciones:

- 1) los L pueden usar el recurso simultáneamente.
- 2) los L y E no pueden usar el recurso al mismo tiempo.

- 3) los E no pueden usar el recurso al mismo tiempo
- 4) los E tienen prioridad sobre los L.
- 5) el recurso solo puede ser concedido a un E , cuando no hay L en ejecución ($le=0$).
- 6) para dar prioridad a los E, establecemos que el recurso solo puede ser concedido a un L cuando no hay E activos ($ea=0$).

El monitor LectoresEscritores que a continuación se presenta cumple con estas condiciones. Este monitor exporta cuatro procedimientos que sincronizan el acceso al recurso: InicioLectura, FinLectura, InicioEscritura, FinEscritura. Los L y E tendrán la siguiente secuencia:

lector :

```

    InicioLectura
        lectura
    FinLectura

```

escritor:

```

    InicioEscritura
        escritura
    FinEscritura

```

DEFINITION MODULE LectoresEscritores;

(x - Módulo para uso de lectores y escritores.

Este módulo trabaja de acuerdo a las siguientes condiciones:

- los escritores tienen prioridad sobre los lectores
- garantiza exclusión mutua entre escritores y entre escritores y lectores
- no existe exclusión mutua entre lectores, es decir varios lectores pueden estar dentro de la misma sección crítica
- un recurso es asignado a un escritor solo cuando los lectores en ejecución son cero ($le=0$)
- para dar prioridad a los escritores, un lector usa el recurso únicamente cuando no existen escritores activos ($ea=0$)

*)

EXPORT QUALIFIED

```
(* proc *) InicioLectura,FinLectura,InicioEscritura,
FinEscritura;
```

PROCEDURE InicioLectura;

(*

- Un lector llama este procedimiento para iniciar la lectura.

*)

PROCEDURE FinLectura;

(*

- Un lector llama este procedimiento cuando ha terminado la lectura.

*)

PROCEDURE InicioEscritura;

(*

```

- Un escritor llama este procedimiento para iniciar
la escritura.

*)

PROCEDURE FinEscritura
(*
-
- Un escritor llama este procedimiento cuando ha
terminado la escritura.

*)

END LectoresEscritores.

IMPLEMENTATION MODULE LectoresEscritores;
FROM Processes IMPORT
    (* tipo *) SIGNAL,
    (* proc *) WAIT,SEND,Init,Awaited;
VAR
    oklectura :SIGNAL;      (* señal para acceso de
                                lectores *)
    okescritura:SIGNAL;    (* señal para acceso de
                                escritores *)
    escritura :BOOLEAN; (* indica si un escritor está
                        en ejecución *)
    ea,le :CARDINAL; (* ea da el número de
                      escritores activos,le da el
                      número de lectores en ejecución
                      *)
PROCEDURE InicioLectura;

```

```
BEGIN
    IF ea<>0 THEN WAIT(oklectura) END;
    INC(ie);
END InicioLectura;

PROCEDURE FinLectura;
BEGIN
    DEC(ie);
    IF ie=0 THEN SEND(okescritura) END;
END FinLectura;

PROCEDURE InicioEscritura;
BEGIN
    INC(ea);
    IF (ie<>0) OR (escritura) THEN WAIT(okescritura) END;
    escritura:=TRUE;
END InicioEscritura;

PROCEDURE FinEscritura;
BEGIN
    escritura:=FALSE;
    DEC(ea);
    IF (ea=0)AND(Awaited(oklectura))THEN SEND(oklectura)
        ELSE SEND(okescritura) END;
END finescritura;

BEGIN (* inicialización de variables *)
    ea:=0;
```

```

le:=0;
escritura:=FALSE;
Init(oklectura);
Init(okescritura)
END LectoresEscritores.

```

A continuación se explica este monitor.

El procedimiento InicioLectura, pregunta si ea (escritores activos) es cero, si es así efectúa un WAIT(oklectura). Esto cumple con la condición 5. Cuando ocurre un SEND(oklectura), el lector continúa e incrementa le (lectores en ejecución) y hace la lectura. Se puede notar que este procedimiento no pregunta nada acerca del estado de otros lectores, ya que entre ellos no se requiere exclusión mutua (condición 1). De esta manera se permite a varios lectores permanecer dentro de la sección crítica (SC).

El procedimiento FinLectura decrementa le y, si le es cero efectúa un SEND(okescritura).

El procedimiento InicioEscritura, primero incrementa ea (es decir establece un nuevo escritor activo aunque posiblemente no pueda entrar a la SC), luego pregunta por la variable escritura o si le es diferente de cero; lo primero es para cumplir con la condición 3 y lo

segundo para la condición 2. Si una de las dos expresiones es verdadera efectúa un WAIT(okescritura) y reiniciará su ejecución luego de un SEND(okescritura). Ahora, esto puede ocurrir en dos lugares: en FinLectura, cuando le es cero (esto reforzaría la condición 4), o en FinEscritura, cuando escritura tiene un valor de falso. Luego de WAIT(okescritura) la variable escritura se asigna a verdadero y el E entra a la SC.

El procedimiento FinEscritura asigna un valor de falso a la variable escritura, decremente ea y efectúa SEND(oklectura) si ea es cero y si existen lectores en espera (con esto evita que un nuevo lector se quede indefinidamente esperando), o de otro modo efectúa SEND(okescritura). Esta última operación reiniciaría a un escritor que haya sido bloqueado en InicioEscritura.

Ahora consideramos un módulo que hace uso del monitor LectoresEscritores.

```
MODULE LectoresEscritoresDemostracion;

(* Este módulo ilustra la sincronización entre lectores
y escritores que comparten un recurso *)

FROM Processes IMPORT
    (* tipo *) SIGNAL,
    (* proc *) StartProcess, WaitMain, Terminate,
```

```
SEND, Init;

FROM LectoresEscritores IMPORT

    (* proc *) InicioLectura,FinLectura,
    InicioEscritura,FinEscritura;

FROM InOut IMPORT

    (* proc *) WriteLn,WriteString,Read;

VAR relevo:SIGNAL; (* señaI para cambio de proceso *)

PROCEDURE lector;
CONST NumeroDeLecturas= ;
VAR lecturas:CARDINAL;
BEGIN
    WriteString('lector ha empezado ');
    WriteLn;
    lecturas:=0;
    LOOP
        InicioLectura;
        WHILE lecturas<NumeroDeLecturas DO
            WriteString('LECTURA ');
            WriteLn;
            INC(lecturas);

            SEND(relevo);
        END;
        FinLectura;
        Terminate;
    END; (* loop *)
END lector;
```

```
PROCEDURE escritor;
CONST NumeroDeEscrituras=  ;
VAR escrituras:CARDINAL;
BEGIN
  WriteString(' escritor ha comenzado ');
  WriteLn;
  escrituras:=0;
  LOOP
    InicioEscrutura;
    WHILE escrituras<NumeroDeEscrituras DO
      WriteString(' ESCRITURA ');
      WriteLn;
      INC(escrituras);
      SEND(releva);
    END;
    FinEscritura;
    Terminate;
  END;(* loop *)
END escritor;

BEGIN (* programa principal *)
  WriteString(' programa principal ha comenzado ');
  WriteLn;
  Init(releva);
  StartProcess(lector,500);
  StartProcess(escritor,500);
  StartProcess(lector,500);
```

```

StartProcess(escriptor,300);

WaitMain;

WriteLn;
WriteString(' programa principal finaliza ');
WriteLn;

END LectoresEscritoresDemostracion.

```

El procedimiento lector puede efectuar tantas lecturas como indica la constante NúmeroDeLecturas; después de cada lectura, efectúa un SEND(relevo), operación que causa los siguientes efectos: el lector permanece en estado listo y el control es pasado a otro proceso en estado listo (que puede ser el programa principal, otro lector o un escritor) y, el lector no abandona su SC, es decir no ejecuta todavía el procedimiento FinLectura. En algún instante el control retornará a este lector, realizará otra lectura y efectuará SEND(relevo), lo cual se repite hasta que haya hecho todas sus lecturas. El lector efectúa FinLectura cuando ha terminado sus lecturas, abandonando así su SC; luego llama al procedimiento Terminate y termina su ciclo.

Similares consideraciones se hacen para los escritores.

A continuación se presenta el resultado del programa, donde se demuestra que se cumple con las condiciones

impuestas para la interacción entre lectores y escritores que comparten un recurso.

'programa principal ha comenzado'

'lector ha comenzado'

LECTURA

'SEND(relevó)'

'escritor ha comenzado'

'WAIT(okescritura)'

LECTURA

'SEND(relevó)'

'lector ha comenzado'

'WAIT(oklectura)'

LECTURA

'SEND(okescritura)'

ESCRITURA

'SEND(relevó)'

'escritor ha comenzado'

'WAIT(okescritura)'

ESCRITURA

'SEND(okescritura)'

ESCRITURA

'SEND(relevó)'

'WaitMain'

ESCRITURA

'SEND(oklectura)'

LECTURA

```
'SEND(relevo)'

LECTURA

'SEND(relevo)'

LECTURA

'SEND(okescritura)'

'programa principal finaliza'
```

Para este programa se considera NúmeroDeLecturas=3 y NúmeroDeEscrituras=2. Las frases entre comillas indican las operaciones que ocurren en ese instante.

Con un programa que contenga

```
StartProcess(lector,500);

StartProcess(lector,500);
```

Se apreciaría que cuando un lector está en su SC otro lector puede entrar a la misma SC.

Observación: en un sistema de quasi-concurrencia por su propia naturaleza, dos procesos nunca efectuarán un intento de entrar a una sección crítica en un mismo instante de tiempo, por esta razón la inclusión de un dígito entre corchetes en el monitor es redundante; en otras palabras no es el monitor en sí quien efectúa exclusión mutua (como sería el caso en un ambiente de concurrencia real); la función de los monitores en las dos secciones precedentes es la de proveer una

abstracción de sincronización para secciones críticas mediante los procedimientos que exporta, de modo que el usuario no intervenga en esa sincronización.

3.3 PROBLEMA DE LOS FILOSOFOS

Si varios procesos están continuamente adquiriendo, usando y liberando uno o más recursos compartidos, necesitamos que ningún proceso quede bloqueado o esperando por una señal que no será enviada; además necesitamos que ningún proceso sea indefinidamente relegado para usar un recurso. El problema de los filósofos puede representar esta situación.

Descripción: hay 5 filósofos que pasan sus vidas ya sea comiendo o pensando, cada uno tiene su propio plato sobre una mesa circular y para comer necesita 2 tenedores, pero solamente hay 5, uno entre cada par de filósofos; los únicos tenedores que un filósofo puede tomar son los que están a su izquierda y derecha. Cada filósofo tiene una estructura idéntica. El problema es simular el comportamiento de los filósofos de forma de evitar bloqueo o 'deadlock' (el requerimiento de un tenedor puede nunca ser concedido) y espera indefinida o 'starvation' (el requerimiento de un tenedor puede ser siempre negado).

De esta descripción podemos señalar los siguientes puntos: filósofos adyacentes no pueden comer al mismo tiempo; con 5 tenedores y la necesidad de dos tenedores para comer, no más de dos filósofos pueden comer simultáneamente.

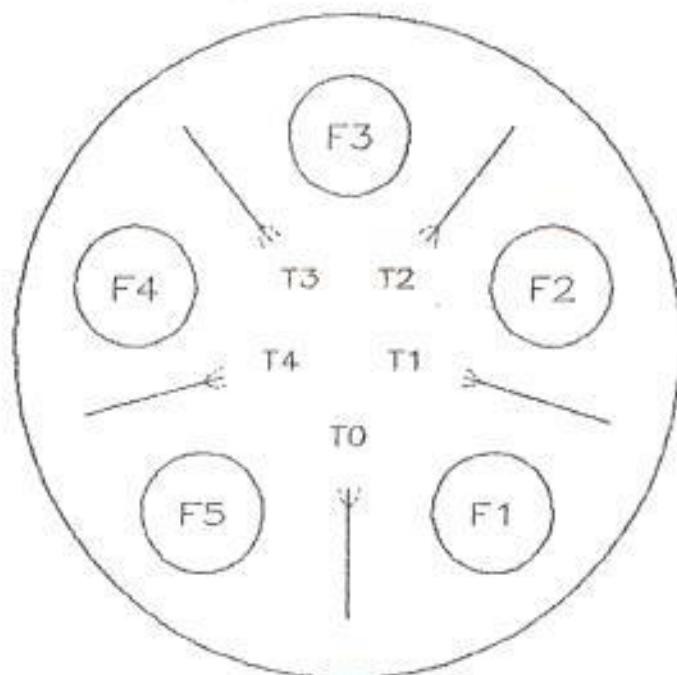


Figura 3.1 Problema de los Filósofos

```

MODULE Filósofos;
(* Programa que resuelve el Problema de los filósofos *)
FROM InOut IMPORT
  (* proc *)WriteString,WriteLn;
```

```

FROM Processes IMPORT
    (* tipo *) SIGNAL,
    (* proc *) WAIT,SEND,StartProcesses,Terminate,
    (* proc *) WaitMain,Init,Awaited;

VAR i:CARDINAL;
    relevo: SIGNAL; (* señal para cambio de proceso *)
    TenedorLibre: ARRAY[0..4] OF BOOLEAN; (* arreglo para
    estado de tenedores *)
    Entrada: ARRAY[i..5] OF SIGNAL; (* arreglo para
    señales de sincronización *)

PROCEDURE TomaTenedor(i:CARDINAL);
(* este procedimiento es usado por un filósofo para
sincronizar el acceso a sus respectivos tenedores *)
BEGIN
    IF NOT ((TenedorLibre [i] MOD (5)) AND (TenedorLibre
    [i-1] MOD (5))) THEN
        WAIT(Entrada[i]);
    END;
    TenedorLibre[i] MOD (5)]:=FALSE;
    TenedorLibre[i-1] MOD (5)]:=FALSE;
END TomaTenedor;

PROCEDURE RetornaTenedor(i:CARDINAL);
(* este procedimiento es usado por un filósofo para
devolver sus respectivos tenedores *)
BEGIN
    TenedorLibre[i] MOD (5)]:=TRUE;
    TenedorLibre[i-1] MOD (5)]:=TRUE;

```

```

TestTenedor;
END RetornaTenedor;

PROCEDURE TestTenedor;
(* este procedimiento es usado por un filósofo para
verificar si existen filósofos en espera de sus
respectivos tenedores *)
VAR i,k,ind:CARDINAL;
    a:ARRAY[1..2]OF CARDINAL;
BEGIN
    j:=1;
    kz:=0;
    WHILE j<=5 DO
        IF ((TenedorLibre [((j)MOD(5))] AND (TenedorLibre [(j-1)MOD(5)])) AND
            (Awaited(Entrada[i]))THEN
            kz:=kz+1;
            ind:=k;
        END; (* if-then *)
        j:=j+1;
    END; (* while *)
    IF kz>0 THEN
        FOR k:=1 TO ind DO
            SEND(Entradafak[k]);
        END; (* for *)
    END; (* if *)

```

```
END TestTenedor;

PROCEDURE filosofo1;
(* procedimiento que representa al filósofo 1 *)
CONST NúmeroDeComidas= ;
VAR ncs:CARDINAL;
BEGIN
  ncs:=0;
  LOOP
    WriteString('filósofo 1 pensando');
    WriteLn;
    TomaTenedor(1);
    WHILE ncs<NúmeroDeComidas DO
      WriteString('filósofo 1 comiendo');
      WriteLn;
      INC(ncs);
      SEND(releva)
    END; (* while *)
    RetornaTenedor(1);
    Terminate;
  END; (* loop *)
END filosofo1;

*
*
*

PROCEDURE filosofo5;
(* procedimiento que representa al filósofo 5 *)
```

```
CONST NúmeroDeComidas= 5;
VAR ncs:CARDINAL;
BEGIN
  ncs:=0;
  LOOP
    WriteString('filósofo 5 pensando');
    writeln;
    TomaTenedor(5);
    WHILE ncs<NúmeroDeComidas DO
      WriteString('filósofo 5 comiendo');
      writeln;
      INC(ncs);
      SEND(relevo);
    END; (* while *)
    RetornaTenedor(5);
    Terminate;
  END; (* loop *)
END filosofo5;
BEGIN
  WriteString('inicio de programa principal');
  writeln;
  Init(relevo);
  FOR i:=1 TO 5 DO
    Init(Entrada[i]);
  END;
  FOR i:=1 TO 4 DO
```

```

TenedorLibre[i]:=TRUE
END;
StartProcess(filosofo1,500);
StartProcess(filosofo2,500);
StartProcess(filosofo3,500);
StartProcess(filosofo4,500);
StartProcess(filosofo5,500);
WaitMain;
WriteString(' fin de programa principal ');
END Filosofos.
```

El programa anterior resuelve el problema de los filósofos en los términos planteados. El módulo PROCESSES utilizado es el mismo de la sección 3.2.

Se declaran dos variables globales que son: TenedorLibre, es un arreglo de cinco elementos de tipo BOOLEANO, donde cada uno representa a un tenedor. Si el valor es verdadero, el tenedor está libre, caso contrario está siendo usado por un filósofo; Entrada, es un arreglo de cinco elementos de tipo SIGNAL. Mediante estas variables sincronizamos el acceso de cada filósofo a los respectivos tenedores, que son los recursos compartidos.

El procedimiento filósofo representa a cada filósofo. La constante NúmeroDeComidas es el número de veces que

como cada filósofo, cada uno de los cuales pasa por el ciclo:

```
REPEAT
    'pensar'
    'comer'
FOREVER;
```

La operación SEND(relevo) causa dos efectos: 1)permite que el filósofo no abandone su sección crítica, es decir no retorne sus respectivos tenedores y 2)permite que otro filósofo empiece o continúe su ejecución.

El procedimiento TomaTenedor es efectuado por cada filósofo después de pensar y cuando desee comer. Este procedimiento verifica si los tenedores adyacentes al plato del correspondiente filósofo están libres, si es así, asigna un valor de falso a cada uno de ellos y procede a comer. Si uno de ellos (o ambos) no está(n) libre(s), es decir tiene un valor de falso, entonces espera por ellos,para lo cual efectúa un WAIT(Entrada[i]). El procedimiento RetornaTenedor asigna un valor de verdadero a cada tenedor y llama al procedimiento TestTenedor. RetornaTenedor es ejecutado por cada filósofo cuando ha terminado su número de comidas. Además ejecuta TestTenedor para permitir a los filósofos en espera de sus respectivos tenedores empezar a comer. El procedimiento TestTenedor como se mencionó

verifica dos cosas: 1) si un filósofo está en espera, es decir haya ejecutado WAIT(Entrada[i]) y 2) si los tenedores del mismo filósofo están libres (o tienen un valor de verdadero). Si se cumplen estas dos condiciones entonces dicho filósofo está habilitado para comer, para lo cual TestTenedor efectúa el correspondiente SEND(Entrada[i]), luego de la cual el filósofo empieza a comer. Pueden existir uno o dos filósofos en esta situación. Nota: el problema de postergación indefinida ('starvation') que puede experimentar un filósofo es resuelto asignando a cada filósofo un cierto número de comidas prefijado, después de los cuales, otro filósofo en espera puede proceder a comer.

Si consideramos la constante NúmeroDeComidas igual a 5 en todos los casos, el siguiente sería el resultado del programa anterior.

```
'inicio de programa principal'  
'filósofo 1 pensando'  
'filósofo 1 comiendo'  
'filósofo 2 pensando'  
'filósofo 1 comiendo'  
'filósofo 3 pensando'  
'filósofo 3 comiendo'  
'filósofo 1 comiendo'  
'filósofo 4 pensando'
```

'filósofo 3 comiendo'
'filósofo 1 comiendo'
'filósofo 5 pensando'
'filósofo 3 comiendo'
'filósofo 1 comiendo'
'filósofo 5 comiendo'
'filósofo 3 comiendo'
'filósofo 5 comiendo'
'filósofo 3 comiendo'
'filósofo 2 comiendo'
'filósofo 5 comiendo'
'filósofo 2 comiendo'
'filósofo 5 comiendo'
'filósofo 2 comiendo'
'filósofo 5 comiendo'
'filósofo 4 comiendo'
'filósofo 2 comiendo'
'filósofo 4 comiendo'
'filósofo 2 comiendo'
'filósofo 4 comiendo'
'filósofo 4 comiendo'
'fin de programa principal'

CAPITULO IV ILUSTRACION DE PROGRAMACION CONCURRENTE Y COMUNICACION ENTRE PROCESOS

4.1 INTRODUCCION

En el capítulo III vimos varios problemas que ilustran lo que es quasi-concurrencia, es decir la existencia de varios procesos que comparten un único procesador. En todos aquellos problemas el procesador era utilizado por cada proceso durante un cierto tiempo que podía variar de acuerdo a la duración de la sección crítica y/o a la extensión del mismo programa. Además la asignación del procesador a cada proceso corría a cargo de los mismos procesos mediante la operación SEND (también en algunos casos por medio de WAIT y Terminate). La operación SEND transfiere el control a un proceso de la cola de procesos listos, de acuerdo a un esquema FIFO (primero en entrar, primero en salir). Como se puede notar, es posible que un proceso retenga el procesador por un lapso prolongado, obligando a otros procesos a permanecer en espera de él.

En el presente capítulo adoptamos otro criterio para la asignación del procesador a los procesos y básicamente consiste en lo siguiente: se crea un proceso especial cuya función será asignar el procesador a cada proceso durante un tiempo fijo; este proceso lo llamamos

repartidor. Cuando el tiempo asignado a cada proceso termina, el control lo toma el repartidor, el cual asigna el procesador a otro proceso para su uso durante el tiempo prefijado a él y así sucesivamente. La asignación del procesador se efectuará de acuerdo al algoritmo 'Round-Robin', el cual se introducirá en la siguiente sección.

En la sección 1.2.1 vimos que un proceso pasa por ciertos estados y que la transición de uno a otro depende de la operación que se efectúe sobre el proceso. La figura 4.1 ilustra los estados de un proceso y las transiciones entre ellos (3).

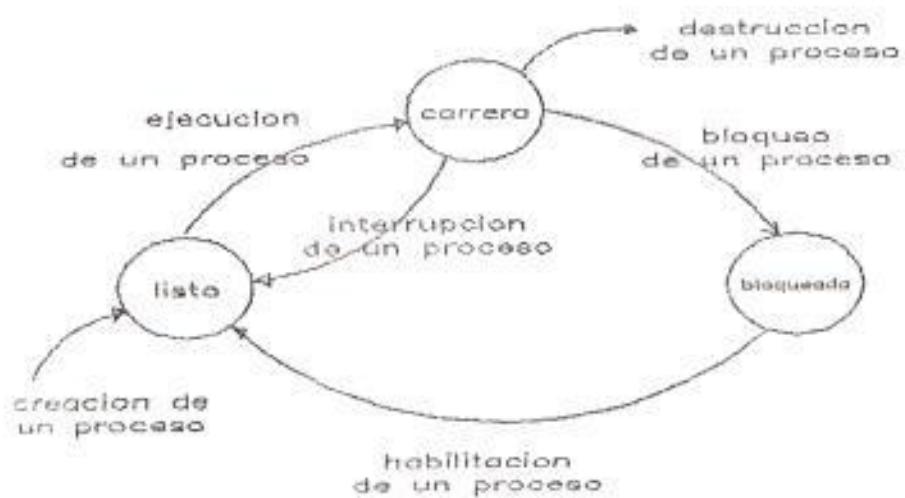


Figura 4.1 Estados de un proceso

Existen otros estados posibles, sin embargo para nuestros fines son suficientes los que se han anotado. Las diferentes operaciones sobre ellos determinan que un proceso cambie de un estado a otro. Por ejemplo: un proceso pasa del estado listo al estado de carrera por medio de una operación realizada por el repartidor o, pasa del estado de carrera al estado bloqueado por medio de una operación que bloquea el proceso, etc. En la sección 4.4 presentaremos un módulo que contiene diversas operaciones para procesos con las cuales ellos pasan de un estado a otro. Este módulo es el manejador de procesos.

4.2 ALGORITMO 'ROUND-ROBIN'

El algoritmo o esquema para la asignación del procesador a los procesos en el presente capítulo se conoce como algoritmo 'Round-Robin' el cual es utilizado en varios sistemas operativos. Los descriptores de los procesos forman una lista circular; un proceso especial es el encargado de asignar el procesador a cada proceso de la lista, este proceso es el repartidor. El método consiste básicamente en asignar el procesador a cada proceso durante un periodo de tiempo fijo el cual se conoce como rebanada de tiempo ('quantum' o 'time

slice'). Cuando el 'quantum' de un proceso termina, el control del procesador es asumido por el repartidor, el cual asigna el procesador al siguiente proceso en estado listo de la cola de procesos, el cual se ejecutará durante un tiempo igual al 'quantum' asignado a él, es decir el repartidor recorre la lista de procesos asignando el procesador por un tiempo fijo. En la sección 4.4 el repartidor es denominado 'Dispatcher'.

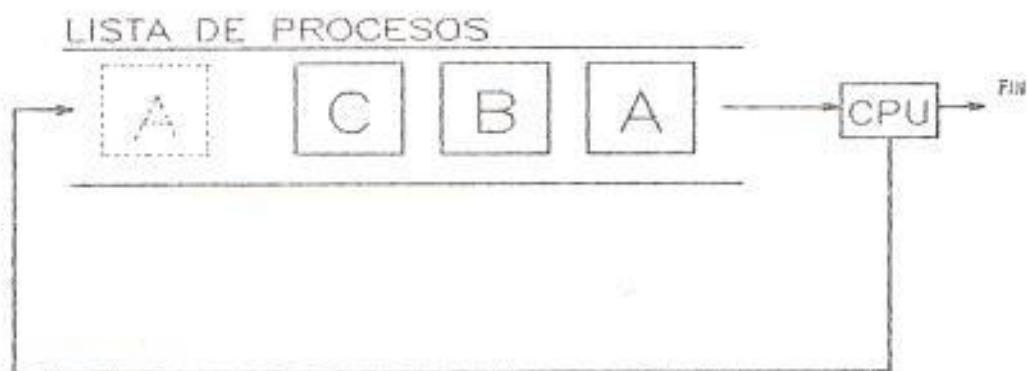


Figura 4.2 Algoritmo 'Round-Robin'

Puede ocurrir dos situaciones: el proceso termina su ejecución antes de que expire su 'quantum' asignado y voluntariamente entregoa el procesador al repartidor el cual lo asigna a otro proceso; o, al final del 'quantum' el proceso es interrumpido, su estado actual es almacenado en la variable de tipo PROCESS de su descriptor de proceso para una posterior re-ejecución y

el control del procesador lo toma el proceso repartidor. Como se aprecia en la figura 4.2, cuando termina el 'quantum' de un proceso, en este caso el A, este es interrumpido y colocado al final de la cola y el repartidor hace correr al proceso que se encuentra al inicio de la cola (siempre que su estado sea listo) en este caso B, luego le tocará el turno al C, etc.; es decir se sigue un esquema FIFO. Sin embargo como veremos en nuestra implantación, la cola es circular y los descriptores de procesos están siempre en la misma posición y el repartidor recorre la lista en busca del siguiente proceso (4).

4.3 CONSIDERACIONES DE TIEMPO

En la sección anterior vimos que a cada proceso se asigna un tiempo fijo para su ejecución o 'quantum', en esta sección veremos el modo de obtener el 'quantum' para cada proceso. Para esto utilizamos el temporizador interno del IBM PC (ver apéndice B) el cual interrumpe el procesador 18.2 veces cada segundo o aproximadamente cada 50 milisegundos; lo que requerimos es entonces elaborar una rutina de servicio para esta interrupción cuyo vector de interrupción es 1CH según las especificaciones dadas por IBM y contar el número de interrupciones para obtener el tiempo para cada proceso.

Como vimos en la sección 2.6, Module-2 provee el procedimiento IOTRANSFER para tratamiento de interrupciones, el cual es como sigue:

PROCEDURE IOTRANSFER (proceso1,proceso2:PROCESS,vector de interrupción:ADDRESS); donde los dos primeros parámetros son de tipo PROCESS y el tercer parámetro es de tipo ADDRESS e indica la dirección del vector de interrupción; proceso 1 representa a la rutina de servicio de interrupción y proceso 2 representa al proceso que será interrumpido. Básicamente funciona de la siguiente manera: cuando se ejecuta IOTRANSFER el control del procesador lo toma el proceso dado por la variable proceso2 y el estado del proceso actual (en este punto la rutina de servicio) es almacenado en la variable proceso1, es decir se produce un TRANSFER(proceso1,proceso2). Cuando ocurre la interrupción el control regresa a la rutina de servicio de esta interrupción en el punto inmediatamente después de IOTRANSFER, en otras palabras la ocurrencia de la interrupción equivale a un TRANSFER(proceso2,proceso1) efectuado en el proceso que es interrumpido.

La estructura básica para la rutina de servicio del temporizador es:

```

PROCEDURE Timer;
  VAR freq:INTEGER;
BEGIN
  LOOP
    freq:=10;
    REPEAT
      IOTRANSFER(timerP,mainP,1CH);
      freq:=freq-1
    UNTIL freq<=0;
    .
    .
  END; (* loop *)
END Timer;

```

Como se observa de este procedimiento, después de cada interrupción la variable 'freq' se decrementa en 1 y luego el control retorna al programa interrumpido por medio de IOTRANSFER; cuando 'freq' es igual a cero ha transcurrido aproximadamente un segundo de tiempo y se tomarán las acciones apropiadas, que como veremos en la siguiente sección consisten en controlar el 'quantum' del proceso actual y si éste es cero se llama al proceso repartidor.

Todas estas consideraciones de tiempo serán incluidas en el módulo manejador de procesos de la siguiente sección.

4.4 MODULO MANEJADOR DE PROCESOS

En esta sección se describe el módulo manejador de procesos conformado por varios procedimientos que efectúan operaciones sobre procesos causando diversos cambios de estado entre ellos, como quedó establecido en la sección 4.1. En este módulo se incluyen también procedimientos para el control de tiempos y tratamiento de la interrupción del temporizador (1).

4.4.1 MODULOS DE DEFINICION E IMPLANTACION

```
DEFINITION MODULE HandlerProcesses;

(* módulo para manejo de procesos. A cada proceso se
asigna un tiempo o 'quantum' para su ejecución *)

EXPORT QUALIFIED

    (* tipo *)SIGNAL,
    (* proc *)SEND,WAIT,Init,Awaited,
        StartTimer,
    ShowProcessDescriptors,PrintProcessDescriptors,
    InstallProcess,TransferToNextProcess,TerminateProcess,
    StartDispatcher;

TYPE SIGNAL;
    (* Variable para sincronización de procesos; es un
puntero a un descriptor de proceso. *)

PROCEDURE SEND(VAR s:SIGNAL);
```

(*Cambia un proceso en estado de espera a estado listo *)

PROCEDURE WAIT(VAR s:SIGNAL);

(* El proceso que efectúa esta operación queda en estado de espera en la cola apuntada por s *)

PROCEDURE Init(VAR s:SIGNAL);

(* Inicializa variables de tipo SIGNAL *)

PROCEDURE Awaited(s:SIGNAL):BOOLEAN;

(* Indica si existen procesos en espera en la cola s *)

PROCEDURE StartTimer(n:CARDINAL);

(* Establece la rutina de servicio de interrupción del temporizador como proceso; n es la frecuencia de ejecución de esta rutina *)

PROCEDURE ShowProcessDescriptors(B:BOOLEAN);

(* Asigna TRUE a B para indicar que se imprime el descriptor de un proceso *)

PROCEDURE PrintProcessDescriptors;

(* Indica por pantalla el contenido de un descriptor de proceso *)

PROCEDURE InstallProcess (P:PROC;Q: CARDINAL;n: CARDINAL;VAR s0:SIGNAL);

(*Crea un proceso y lo incluye en la lista de procesos creados. P es el procedimiento creado como proceso, Q es el 'quantum' para el proceso, n es el espacio de memoria asignado al proceso, s0 es un puntero al proceso *)

PROCEDURE TransferToNextProcess;

(* Transfiere el control del procesador al siguiente

```

proceso listo *)
PROCEDURE TerminateProcess;
(* Termina un proceso. Su descriptor es excluido de la
lista *)
PROCEDURE StartDispatcher;
(* Establece el proceso repartidor ('Dispatcher') como
proceso. Su descriptor es incluido en la lista de
procesos creados *)
END HandlerProcesses.

IMPLEMENTATION MODULE HandlerProcesses;
FROM SYSTEM IMPORT
    (* tipo *) ADDRESS,PROCESS,
    (* proc *) NEWPROCESS, TRANSFER, IOTRANSFER, ADR,
    SIZE,TSIZE;
FROM Devices IMPORT
    (* proc *) SaveInterruptVector, RestoreInterruptVector;
    GetDeviceStatus,SetDeviceStatus;
FROM Storage IMPORT
    (* proc *) ALLOCATE,DEALLOCATE;
FROM Input IMPORT
    (* proc *) WriteCard,WriteLn,WriteString;
FROM System IMPORT
    (* proc *) TermProcedure;
CONST
    intvectnum=ICH;      (* dirección del vector de
    interrupción del temporizador *)

```

device=0; (* número del bit correspondiente al dispositivo (en este caso la salida cero del 8253) en el registro del controlador de interrupciones 8259 *)

TYPE

```

  SIGNAL ^POINTER TO ProcessDescriptor;
  ProcessDescriptor=RECORD
    Number:CARDINAL; (* identificación de proceso *)
    Next:SIGNAL; (* lista de procesos *)
    Queue:SIGNAL; (* cola de procesos en espera *)
    Cor:PROCESS; (* registros del proceso *)
    Ready:BOOLEAN; (* estado del proceso *)
    Quantum:CARDINAL; (* tiempo del proceso *)
    Wsp:ADDRESS; (* dirección del proceso *)
    N:CARDINAL (* espacio de trabajo *)
  END;

```

VAR

```

  cp:SIGNAL; (* apunta al proceso actual *)
  np:SIGNAL; (* apunta al siguiente proceso *)
  dp:SIGNAL; (* apunta al proceso repartidor *)
  TotalProcessCount:CARDINAL; (* número de procesos creados *)
  ActiveProcessCount:CARDINAL; (* número de procesos en la lista *)
  TimeLeft:INTEGER; (* tiempo de cada proceso *)
  ShowPD:BOOLEAN; (* para indicar por pantalla el

```

```

descriptor *)
timerP:PROCESS; (* estado de rutina del reloj *)
mainP :PROCESS; (* estado del proceso actual *)
count:CARDINAL; (* frecuencia de ejecución de 'Timer'
*)
oldintvector:ADDRESS;      (*      para      procedimiento
SaveInterruptVector *)
oldDeviceStatus:BOOLEAN;   (*      para      procedimientos
'GetDeviceStatus' y 'SetDeviceStatus' *)
active:BOOLEAN; (* indica si 'Timer' ha sido instalado
como proceso *)

PROCEDURE Timer;
VAR freq:CARDINAL;
BEGIN
LOOP
freq:=count;
REPEAT
IOTRANSFER(timerP,mainP ,intvectnum);
freq:=freq-1;
UNTIL freq<=0;
DEC(TimeLeft);
IF Timeleft <= 0 THEN FindDispatcherEndOfQuantum END;
END;
END Timer;
PROCEDURE StartTimer (n:CARDINAL);
BEGIN

```

```
count:=n;

IF NOT active THEN
  SaveInterruptVector(intvectnum, oldintvector);
  GetDeviceStatus(device, oldDeviceStatus);
  active:=TRUE;
  NEWPROCESS(Timer,    ADR(workspace),    SIZE(workspace),
  timerP);
  TRANSFER (mainP, timerP);
  GetDeviceStatus(device, TRUE);
END;

END StartTimer;

PROCEDURE StopTimer;
BEGIN
  IF active THEN
    active:=FALSE;
    SetDeviceStatus(device, oldDeviceStatus);
    RestoreInterruptVector(intvectnum, oldintvector);
  END;
END StopTimer;

PROCEDURE ShowProcessDescriptors(B:BOOLEAN);
BEGIN
  ShowPD:=B;
END ShowProcessDescriptors;

PROCEDURE PrintProcessDescriptors;
VAR s0:SIGNAL;
BEGIN
```

```

WriteLn;
WriteLn;
s0:=cp;
REPEAT
  WITH s0^ DO
    WriteString(' num. de proc.: ');
    WriteCard(Number,5);
    WriteString('estado: ');
    IF Ready THEN WriteString(' lista')
      ELSE WriteString(' bloquedo');
    END (* if-then-else *)
    WriteString(' ');
    WriteString(' Quantum ');
    WriteCard(Quantum,5);
    WriteLn;
  END; (* with *)
  s0:=s0^.Next;
UNTIL s0=cp;
END PrintProcessDescriptors;

PROCEDURE      FindNextProcess(VAR          pdPtr: SIGNAL; VAR
np: SIGNAL); :;

BEGIN
  np:=pdPtr;
  REPEAT      np:=np^.Next      UNTIL      np^.Ready      AND
  (np^.Quantum>0);
  IF  np=cp  THEN  WriteString(' No existen procesos

```

```

listos. Bloqueo');

HALT;END;

pdPtr:=np;
END FindNextProcess;

PROCEDURE
InstallProcess(P:PROC;Q:CARDINAL;n:CARDINAL;VAR
s0: SIGNAL);
VAR wsp: ADDRESS;
BEGIN
  ALLOCATE(wsp,n);
  ALLOCATE(s0,TSIZE(ProcessDescriptor));
  INC(TotalProcessCount);
  WITH s0^ DO
    Number:=TotalProcessCount;
    Next:=cp^.Next;
    Dueue:=NIL;
    Ready:=TRUE;
    Quantum:=0;
    Wsp:=wsp;
    N:=n;
  END;
  cp^.Next:=s0;
  NEWPROCESS(P,wsp,n,s0^.Cor);
  IF ShowPD THEN PrintProcessDescriptors END;
END InstallProcess;

PROCEDURE SEND(VAR s: SIGNAL);

```

```

VAR s0: SIGNAL;
BEGIN
  IF s<>NIL THEN
    s0:=s;
    WITH s0^ DO
      s1=Queue;
      Ready:=TRUE;
      Queue:=NIL;
    END;
  END;
END SEND;

PROCEDURE WAIT( VAR s: SIGNAL );
VAR s0,s1: SIGNAL;
BEGIN
  IF s=NIL THEN s1:=cp;
  ELSE
    s0:=s;
    s1:=s0^.Queue;
    WHILE s1<>NIL DO
      s0:=s1;
      s1:=s0^.Queue;
    END;
    s0^.Queue:=cp;
  END;
  cp^.Ready:=FALSE;
  np:=NIL;
END;

```

```
EndQuantum;

END WAIT;

PROCEDURE Awaited(s:SIGNAL):BOOLEAN;
BEGIN
  RETURN s<>NIL;
END Awaited;

PROCEDURE Init(VAR s:SIGNAL);
BEGIN
  s:=NIL;
END Init;

PROCEDURE StartQuantum(n:CARDINAL);
BEGIN
  TimeLeft:=n;
  dispatch
END StartQuantum;

PROCEDURE dispatch;
VAR so:SIGNAL;
BEGIN
  cp^.Ready:=FALSE;
  so:=cp;
  cp:=nps;
  TRANSFER(so^.Cor, cp^.Cor)
END dispatch;

PROCEDURE EndQuantum;
BEGIN
  TimeLeft:=0;
END;
```

```

FindDispatcher
END EndQuantum;

PROCEDURE FindDispatcher;
VAR so:SIGNAL;
BEGIN
  dp^.Ready:=TRUE;
  so:=cp;
  cp:=dp;
  TRANSFER(so^.Cor,co^.Cor)
END FindDispatcher;

PROCEDURE FindDispatcherEndQuantum;
BEGIN
  dp^.Ready:=TRUE;
  np:=NIL;
  cp:=dp;
  mainP:=cp^.Cor
END FindDispatcherEndQuantum;

PROCEDURE Dispatcher;
VAR pdPtr:SIGNAL;
BEGIN
  np:=NIL;
  pdPtr:=dp;
  LOOP
    IF np=NIL THEN FindNextProcess(pdPtr,np)END;
    StartQuantum(np^.Quantum);
  END;
END Dispatcher;

```

```

PROCEDURE StartDispatcher;
VAR s0: SIGNAL;
BEGIN
  InstallProcess(Dispatcher,0,500,dp);
  s0:=cp;
  cpt:=dp;
  TRANSFER(s0^.Cor,cpt^.Cor);
END StartDispatcher;

PROCEDURE TransferToNextProcess;
BEGIN
  np:=NIL;
  EndQuantum;
END TransferToNextProcess;

PROCEDURE TerminateProcess;
VAR s0: SIGNAL;
BEGIN
  DEC(ActiveProcessCount);
  cp^.Ready:=FALSE;
  s0:=cp;
  REPEAT cp:=cp^.Next UNTIL cp^.Next=s0;
  cp^.Next:=s0^.Next;
  ALLOCATE(cp,TSIZE(ProcessDescriptor));
  cp^.Cor:=s0^.Cor;
  DEALLOCATE(s0^.Nsp,s0^.N);
  IF ActiveProcessCount <>2
    THEN np:=NIL
  ELSE np:=dp^.Next END;

```

```

EndQuantum

END TerminateProcess;

BEGIN (* inicialización *)
  ShowPD:=FALSE;
  TotalProcessCount:=1;
  ActiveProcessCount:=1;
  ALLOCATE(cp,TSIZE(ProcessDescriptor));
  WITH cp^ DO
    Number:=1;
    Next:=cp;
    Ready:=TRUE;
    Queue:=NIL;
    Quantum:=0
  END;
  active:=FALSE;
  TermProcedure(StopTimer);
END HandlerProcesses.

```

4.4.2 DESCRIPCION DE LOS MODULOS

El módulo 'HandlerProcesses' exporta un tipo de dato y varios procedimientos. El tipo de dato SIGNAL es un puntero a un descriptor de proceso. Los procedimientos utilizan variables de este tipo para diversas operaciones sobre procesos, accediendo a sus descriptores a través de variables de tipo SIGNAL.

Cada proceso está representado por un bloque de información denominado 'ProcessDescriptor' o descriptor de proceso que contiene varios campos relativos al los procesos que son:

Number: asigna un número de identificación para el proceso de acuerdo al orden de creación.

Next : este campo que es de tipo SIGNAL es utilizado para formar la lista de procesos creados.

Queue: este campo que es de tipo SIGNAL es utilizado para formar la cola de procesos en espera.

Cor : en este campo que es de tipo PROCESS se guarda el estado de los registros de máquina del proceso cuando es creado con NEWPROCESS, o cuando es suspendido para su posterior reinicio.

Ready : indica el estado del proceso. Si es verdadero el proceso está en estado listo o en estado de carrera. Si es falso está en estado de espera en alguna cola.

Quantum: este campo indica el tiempo asignado al proceso para uso del procesador.

Wsp : contiene la dirección del espacio de memoria para el proceso.

N : contiene el tamaño de memoria asignado para trabajo del proceso.

Mediante el procedimiento 'InstallProcess' se forma una

lista circular de descriptores de proceso; esta lista será recorrida por el repartidor ('Dispatcher') como se indicó en la sección 4.1, con el fin de asignar el procesador a cada proceso en estado listo para lo cual utiliza el algoritmo 'Round-Robin'.

Hay tres variables globales de tipo SIGNAL, las cuales son usadas por varios procedimientos del módulo.

cp : apunta en todo momento al descriptor de proceso que se está ejecutando, denominado proceso actual.

np : apunta al siguiente proceso en ejecutarse; el proceso repartidor ('Dispatcher') transfiere el control al proceso cuyo descriptor es apuntado por np. Previamente el procedimiento 'FindNextProcess' encuentra el siguiente proceso listo que va a ejecutarse y al descriptor de él apunta con np.

dp : este puntero en todo momento este apuntando al descriptor del proceso repartidor, lo que permite la localización de este proceso en dos circunstancias: cuando termina el 'quantum' de un proceso o cuando un proceso suspende su ejecución (o se autoelimina); dp apunta al repartidor desde que se ejecuta 'StartDispatcher'.

TotalProcessCount es incrementada por el procedimiento 'InstallProcess' e indica el número de procesos

presentes en la lista.

ActiveProcessCount: es incrementada por 'InstallProcess' y decrementada por 'TerminateProcess' e indica el número de procesos que quedan en la lista.

ShowPD: si su valor es verdadero se ejecuta el procedimiento 'PrintProcessDescriptors'.

TimeLeft: esta variable sirve para el control del tiempo asignado a cada proceso; el procedimiento 'StartQuantum' asigna a esta variable el valor del 'quantum' del siguiente proceso a ejecutarse. Es decrementada en la rutina del temporizador, cuando llega a cero (el 'quantum' ha terminado) el proceso actual es suspendido y el repartidor es llamado.

Las siguientes variables son usadas por la rutina de servicio del temporizador:

WorkSpace : espacio asignado para el procedimiento 'Timer'.

timerP : esta variable de tipo PROCESS guarda el estado del procedimiento 'Timer'.

mainP : esta variable de tipo PROCESS guarda el estado del proceso actual. Despues de cada IOTRANSFER en 'Timer' el proceso actual representado por 'mainP' reinicia su ejecución.

count : representa la frecuencia de ejecución de la

rutina de servicio, es decrementada cada vez que ocurre una interrupción del temporizador, cuando llega a cero se verifica el valor de 'TimeLeft'. Es decir el 'quantum' de cada proceso es un múltiplo entero del valor inicial de esta variable.

Los procedimientos del módulo son :

Timer : es la rutina de servicio del temporizador. Controla el 'quantum' de los procesos, si es cero llama al repartidor por medio de 'FindDispatcherEndQuantum'.

StartTimer: establece al procedimiento 'Timer' como proceso mediante NEWPROCESS.

StopTimer: para terminación del procedimiento que atiende la interrupción. Este procedimiento es usado conjuntamente con 'TermProcedure' del módulo 'System' (5).

PrintProcessDescriptors: indica por pantalla el contenido de los descriptores de proceso.

SEND(VAR s:SIGNAL): convierte en listo al primer proceso en espera de la cola apuntada por s. El proceso que llama este procedimiento continúa su ejecución.

WAIT(VAR s: SIGNAL): el proceso que llama este procedimiento se inserta al final de la cola apuntada por s, de donde es removido por medio de un SEND; hace

la asignación 'np:=NIL' y llama al repartidor por medio de 'EndQuantum'.

Init(VAR s:SIGNAL): inicializa variables de tipo SIGNAL.

Awaited(s:SIGNAL): BOOLEAN: cuando es verdadero indica que hay al menos un proceso en la cola de s.

InstallProcess(P:PROC;Q:CARDINAL;n:CARDINAL;VAR s0:SIGNAL): crea un nuevo proceso y su descriptor lo incluye en la lista de procesos creados; el estado de este proceso es de listo. P es el procedimiento que va a ejecutarse como corriputina, Q es el 'quantum' asignado a la corriputina, n es el espacio de trabajo, s0 es un puntero que apunta al descriptor del proceso creado y por medio del cual se accede al descriptor.

TerminateProcess: es llamado por un proceso cuando requiere autoeliminarse, su descriptor es excluido de la lista. Hace la asignación 'np:=NIL' y llama al repartidor por medio de 'EndQuantum'. Cuando todas las corriputinas, excepto el repartidor y el programa principal, han terminado ('ActiveProcessCount=2') el control es transferido al programa principal.

FindNextProcess: encuentra al siguiente proceso que va a ejecutarse al cual apunta con el puntero np, este proceso deberá estar en estado listo y tener 'quantum' mayor que cero.

TransferToNextProcess: transfiere el control al siguiente proceso listo, para lo cual llama al repartidor por medio de 'EndQuantum'.

Dispatcher: este proceso como ya se ha indicado es el encargado de asignar el procesador a las diferentes corutinas, llama a 'FindNextProcess' para hallar el siguiente proceso a ejecutarse el cual es apuntado por np, luego ejecuta 'StartQuantum' el cual transfiere el control al nuevo proceso.

StartQuantum(n:CARDINAL): este procedimiento es llamado por el repartidor para transferir el control al nuevo proceso, esto lo hace por medio del procedimiento 'dispatch'.

EndQuantum: este procedimiento es llamado por las corutinas cuando requieren autosuspenderse, asigna cero al 'quantum' del proceso actual y llama a 'FindDispatcher'.

dispatch: transfiere el control al proceso apuntado por np y a su descriptor apunta con el puntero cp.

FindDispatcher: transfiere el control al proceso repartidor el cual ha sido suspendido previamente .

FindDispatcherEndQuantum: este procedimiento invoca al proceso repartidor cuando termina el 'quantum' de una

corrutina, asigna a la variable 'mainP' el estado del proceso repartidor para que este reanude su ejecución por medio del IOTRANSFER de 'Timer'.

StartDispatcher: establece como corrutina al repartidor y lo incluye en la lista de procesos creados para lo cual ejecuta 'InstallProcess(dispatcher,0,500,dp)', con este operación le asigna un 'quantum' de cero, un espacio de memoria de 500 BYTES y dp apunta a su descriptor a partir de este momento. Por último transfiere el control al repartidor.

En la parte de inicialización se crea un descriptor de proceso (por medio del procedimiento ALLOCATE del módulo Storage) que representa al programa principal; el 'quantum' de este proceso es cero y es apuntado por la variable cp, en el campo 'Number' se asigna 1, es decir este es el primer proceso creado, el campo 'Ready' es TRUE. 'TotalProcessCount' y 'ActiveProcessCount' toman el valor de 1. Además se ejecuta 'TermProcedure(StopTimer)', operación que instala 'StopTimer' como una rutina de terminación, con el fin de parar correctamente el procedimiento que atiende la interrupción del temporizador cuando el programa que lo usa termina (5).

4.5 DEMOSTRACION DEL MODULO MANEJADOR DE PROCESOS

El módulo 'DemostraciónHandlerProcess' importa procedimientos del manejador de procesos a base de los cuales ejecuta varios procesos quasi-concurrentemente, siendo el 'quantum' de cada uno de ellos igual o mayor que un segundo de tiempo aproximadamente.

```

MODULE DemostraciónHandlerProcesses;

(* módulo que ejecuta varios procesos en quasi-
concurrencia. Importa procedimientos de
'HandlerProcesses'. Cada proceso tiene un 'quantum' de
tiempo para su ejecución.*)

FROM InOut IMPORT

    (* proc *) WriteString,WriteLn,ReadCard;

FROM HandlerProcesses IMPORT

    (* proc *) StartTimer,ShowProcessDescriptors,
    InstallProcess,TerminateProcess,Start-
    Dispatcher;

VAR l:CARDINAL;

PROCEDURE P1;
CONST iter1= ;

    iter2= ;

VAR i,j:CARDINAL;

```

```
BEGIN
  LOOP
    FOR i:=1 TO iter1 DO
      FOR j:=1 TO iter2 DO
        (* *)
      END;
      WriteString(' proceso 1 en iteración ');
      writeln;
    END; (* for *)
    TerminateProcess;
  END; (* loop *)
END P1;

PROCEDURE P2;
CONST iter1= ;
      iter2= ;
VAR i,j:CARDINAL;
BEGIN
  LOOP
    FOR i:=1 TO iter1 DO
      FOR j:=1 TO iter2 DO
        (* *)
      END;
      WriteString(' proceso 2 en iteración ');
      writeln;
    END; (* for *)
    TerminateProcess;
```

```

END; (* loop *)
END P2;

PROCEDURE P3;
CONST iter1= ;
      iter2= ;
VAR i,j:CARDINAL;
BEGIN
  LOOP
    FOR i:=1 TO iter1 DO
      FOR j:=1 TO iter2 DO
        (* *)
      END;
      WriteString(' proceso 3 en iteración ');
      WriteLn;
    END; (* for *)
    TerminateProcess;
  END; (* loop *)
END P3;

BEGIN (* programa principal *)
  ShowProcessDescriptors(TRUE);
  WriteString(' entre quantum para P1 en segundos: ');
  ReadCard(i);
  WriteLn;
  InstallProcess(P1,i,500,s0);
  WriteString(' entre quantum para P2 en segundos: ');

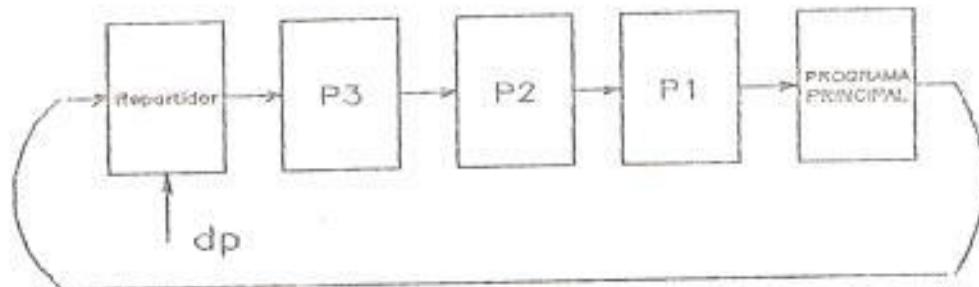
```

```

ReadCard(i);
WriteLn;
InstallProcess(P2,i,500,s0);
WriteString(' entre quantum para P3 en segundos: ');
ReadCard(i);
InstallProcess(P3,i,500,s0);
StartTimer(18);
StartDispatcher;
WriteString('todos los procesos han terminado ');
WriteLn;
END DemostraciónHandlerProcesses.

```

La lista de descriptores de proceso de este módulo se visualiza en el siguiente gráfico.



El programa principal ejecuta al final 'StartDispatcher' lo que transfiere el control al repartidor y además el programa principal queda en espera (con su estado guardado en el campo 'Cor' de su descriptor) de la

finalización de todas las corutinas; cuando esto ocurre el programa principal es reiniciado y se imprime el mensaje: 'todos los procesos han terminado'. El repartidor busca el primer proceso, que como se ve en el gráfico es P3 y este proceso entra en ejecución durante el tiempo asignado, luego del cual el repartidor asume el control y asigna el procesador a P2 ,etc. Los procesos terminan por invocación al procedimiento 'TerminateProcess'. Cuando todos han terminado, el control regresa al programa principal inmediatamente después de 'StartDispatcher'.

El resultado que se vería en pantalla es el siguiente.

```
proceso 3 en iteración
proceso 2 en iteración
proceso 1 en iteración
proceso 3 en iteración
proceso 2 en iteración
proceso 1 en iteración
proceso 3 en iteración
:
:
:
todos los procesos han terminado
```

4.6 OBSERVACIONES GENERALES

- 1) La rutina del reloj se ejecuta 18 veces cada segundo aproximadamente, es decir que si asignamos a la variable 'freq' un valor de 18 tenemos que el 'quantum' de cada proceso es verificado una vez por segundo, para lo cual se decremente la variable 'TimeLeft' (a la que se asigna un valor correspondiente al 'quantum' del proceso en el procedimiento 'StartQuantum'), si esta variable es cero, el 'quantum' ha terminado y se llama al repartidor.
- 2) Con un 'StartTimer(18)' en el programa de demostración del manejador de procesos, contabilizamos el 'quantum' de cada proceso en segundos; de esta manera el 'quantum' mínimo sería de 1 segundo. Con un argumento menor a 18 podemos obtener un 'quantum' menor a 1 segundo de tiempo.
Podemos hacer varias observaciones a partir de la figura 4.1 y de los procedimientos del módulo manejador de procesos 'HandlerProcesses':
- 3) El estado de un proceso lo determina el campo 'Ready' del descriptor respectivo, si es verdadero el proceso está en estado listo, caso contrario está en estado de espera o bloqueado; si está en

estado de carrera 'Ready' tambien es verdadero.

- 4) El procedimiento 'InstallProcess' crea un proceso mediante NEWPROCESS, su descriptor de proceso lo inserta en la lista de procesos creados y le asigna el estado listo por medio de la instrucción: 'sonReady:=TRUE'.
- 5) La operación que produce un cambio del estado listo al estado de carrera la efectúa el procedimiento 'Dispatcher'.
- 6) La operación que produce un cambio del estado de carrera al estado de bloqueo o espera la efectúa el procedimiento WAIT.
- 7) La operación que produce una transición del estado de espera al estado listo es efectuada por el procedimiento SEND.
- 8) La operación que produce un cambio del estado de carrera al estado listo se efectúa al final del 'quantum' del proceso.
- 9) El procedimiento 'TerminateProcess' elimina un proceso cuando este está en estado de carrera.
- 10) Todas la transiciones son llevadas a cabo por el procedimiento TRANSFER que guarda el estado del

proceso que es suspendido para su posterior reinicio.

- 11) Las únicas transiciones iniciadas por el propio proceso son mediante los procedimientos WAIT y 'TerminateProcess', las otras transiciones de estado son realizadas por eventos externos al proceso.
- 12) El procedimiento SEND del módulo 'HandlerProcesses' es diferente al SEND del módulo PROCESSES; en el segundo caso, SEND hace una transferencia a un proceso, en el primer caso SEND no efectúa ninguna transferencia, lo que hace es convertir en listo a un proceso en estado de espera y el proceso que llamó a SEND continúa hasta que termina su 'quantum'.

CONCLUSIONES Y RECOMENDACIONES

CONCLUSIONES

1. En el capítulo I se presentaron algunos conceptos básicos de programación concurrente, entre ellos constan: procesos, semáforos, monitores y vimos además que estos pueden ser implantados con las herramientas suministradas por Modula-2. Por ejemplo un proceso en Modula-2 se define mediante el procedimiento NEWPROCESS y el tipo PROCESS, tal como quedó demostrado en el procedimiento 'StartProcess' del módulo PROCESSES. Un semáforo es implantado mediante la variable tipo SIGNAL, que como vimos es un puntero a un descriptor de proceso, y las operaciones WAIT y SEND de un semáforo son efectuadas sobre variables de tipo SIGNAL. Para monitores, Modula-2 dispone de los módulos de biblioteca (Definición e Implementación) o de módulos internos. Estos módulos se convierten en monitores añadiendo un dígito en un rango de 0 a 7 encerrado entre corchetes junto al nombre del módulo (5).
2. Las herramientas de sincronización analizadas en esta tesis son: semáforos, regiones críticas y monitores. Cuando se usan semáforos es

responsabilidad del programador establecer las señales correctas para sincronizar procesos; en cuanto a regiones críticas y monitores, es el compilador el encargado y responsable de efectuar exclusión mutua entre procesos. Se debe indicar que en quasi-concurrencia es innecesario añadir un dígito a la declaración de módulos para formar monitores, porque por definición nunca dos o más procesos intentarán acceder a él (al monitor) a un mismo tiempo (6).

3. Modula-2 posee propiedades para trabajar con quasi-concurrencia. Entre estas características podemos mencionar: el tipo PROCESS, los procedimientos NEWPROCESS, TRANSFER e IOTRANSFER, todos los cuales son exportados del módulo SYSTEM que viene incluido dentro del compilador de Modula-2.
4. El módulo PROCESSES, provee una abstracción de comunicación y sincronización entre procesos y soporta quasi-concurrencia, como se ilustró en el capítulo III. Este módulo puede ser modificado de acuerdo al problema en particular, por ejemplo para el problema productor/consumidor se incluyó un contador de señales para mantener el registro de los datos depositados y removidos del 'buffer'. En el problema lectores/escritores esto no es

necesario y el acceso a la sección crítica se lo hace de acuerdo a otras condiciones del problema (7).

5. Al módulo PROCESSES básico (sección 2.4.5) se añadió los procedimientos 'Terminate' y 'WaitMain'. 'Terminate' termina un proceso y 'WaitMain' suspende el programa principal en una cola (la de la señal 'idle'). Es decir la función del programa principal es iniciar todos los procesos, luego de lo cual espera en dicha cola la terminación de todos ellos. Es decir, cada proceso puede eliminarse por sí mismo sin afectar el funcionamiento de los otros; el último proceso en llamar a 'Terminate' rehabilita el programa principal, suspendido hasta ese momento en la cola de la señal 'idle'.
6. El esquema utilizado en el capítulo III para la asignación del procesador a los procesos es el denominado FIFO (primero en entrar, primero en salir). Este esquema es implantado internamente por los procedimientos WAIT y SEND (aunque también por 'Terminate'). De esta manera se logra un sistema equitativo para servir a los procesos de acuerdo a su orden de llegada.

7. En lo que respecta a los problemas abordados en el capítulo III, se concluye que son los mismos procesos los que asignan el procesador a otro proceso y lo hacen básicamente mediante una operación SEND. También las operaciones WAIT y 'Terminate' hacen una asignación a otro proceso, la primera cuando el proceso no puede entrar a una sección crítica y la segunda cuando el proceso termina.
8. El procedimiento SEND en el módulo PROCESSES efectúa una transferencia a un proceso, es en esta operación únicamente en donde tiene lugar una transferencia de procesador, debido a que son los mismos procesos los que asignan el procesador. Los procesos manejados por el módulo 'HandlerProcesses' del capítulo 4 no asignan el procesador, sino que es el repartidor el que lo hace, de modo que incluir una operación de transferencia (TRANSFER) en el procedimiento SEND de este módulo parece innecesario.
9. Para la solución de los problemas del capítulo 3, se ha usado una de las propiedades de Modula-2: la capacidad de implantar 'abstracción de datos'. Es decir construir módulos que exportan tipos de datos y procedimientos sobre estos datos. Como ejemplos podemos mencionar el módulo PROCESSES que provee una abstracción de sincronización entre procesos,

el módulo BufferCircular que provee una abstracción de comunicación entre procesos.

10. El módulo 'HandlerProcesses' es un módulo básico para manejo de procesos; provee operaciones para procesos que permiten un cambio de estado, creación de procesos nuevos, terminación de procesos y también un modo de asignar un tiempo de ejecución a cada proceso de un sistema de quasi-concurrencia; esta última característica lo diferencia del módulo PROCESSES.

RECOMENDACIONES

1. El objetivo de este trabajo, que es uno de los primeros basados en el lenguaje Modula-2, es motivar el estudio de programación concurrente conjuntamente con este lenguaje que está siendo usado actualmente en el diseño de sistemas operativos y software de control de procesos, para lo cual posee las facilidades necesarias tal como ha quedado señalado en esta tesis. En este sentido el presente trabajo ilustra varios problemas pertenecientes a programación concurrente, a partir de los cuales se pueden iniciar trabajos de mayor complejidad (B).
2. Elaborar un monitor para el 'buffer' del problema productor-consumidor que utilice variables para sincronización sin contador de señales; en el 'buffer' explicado en la sección 3.1 si es necesario el contador de señales y puede ser usado por varios productores y consumidores. Además con respecto al problema de lectores y escritores se puede solucionar bajo condiciones de prioridad a lectores, o bajo condiciones de acuerdo a la aplicación del problema.

3. Además del problema productor-consumidor que ilustra cómo dos o más procesos se pueden comunicar entre sí por medio de una variable común, investigar otro tipo de comunicación entre procesos que se implante por medio de algún protocolo de comunicación o que permita a dos procesos pertenecientes a un sistema formado por muchos procesos, comunicarse entre sí; a este respecto existe un tipo de comunicación denominado CHANNEL que se utiliza para la comunicación entre dos procesos (2).
4. Crear un monitor para secciones críticas con los procedimientos SEND y WAIT del módulo 'HandlerProcesses'; al monitor se le asigna una prioridad y al módulo 'HandlerProcesses' también se asigna una prioridad; si la prioridad del monitor es mayor, entonces el proceso no es interrumpido cuando está dentro de su sección crítica.
5. La sección crítica sería una variable compartida declarada como monitor. Es decir los procedimientos del monitor serían los únicos que acceden a la variable; este monitor debería tener la más alta prioridad para no ser interrumpido por la interrupción del temporizador. Esta sería una manera de evitar interrupciones dentro de secciones críticas y de asegurar exclusión mutua.

- 6. El sistema provee varios módulos para comunicación por medio de la puerta asíncrona serie RS232. Uno de estos módulos es el 'RS232Int' que exporta procedimientos para leer y escribir datos en dicha puerta y para atender interrupciones externas por la misma puerta; podemos construir módulos con estos procedimientos para intercambiar información con un dispositivo externo, por ejemplo otro microcomputador.

BIBLIOGRAFIA

1. Brinch Hansen P., *Operating Systems Principles* (1ra. edición; Englewood Cliffs: Prentice Hall, 1973)
2. Ford G., Wiener R., *Modula-2: A Software Development Approach* (John Wiley & Sons, 1985)
3. Deitel Harvey M., *An Introduction to Operating Systems* (1ra. edición; Reading, Massachusetts: Addison Wesley, 1984)
4. Peterson J.L., *Operating Systems Concepts* (2da. edición; Reading, Massachusetts: Addison Wesley, 1985)
5. Logitech Inc., *Modula-2/86*, 1985
6. Wirth Niklaus, *Programming in Modula-2* (3ra. edición; Harrisonburg, Virginia: R.R. Donnelley and Sons, 1985)
7. Ben-Ari M., *Principles of Concurrent Programming* (1ra. edición; Englewood Cliffs: Prentice Hall, 1982)
8. Real-Time Computer Control (1ra. edición; London: S. Bennet and D.A. Linkens, 1984)
9. Sargent M., Shoemaker R.L., *The IBM Personal Computer* (1ra. edición; Reading, Massachusetts: Addison Wesley, 1985)
10. Wilson C.R., "Coprocessing in Modula-2", *Revista BYTE*, ABRIL 1985,
pp. 113-117