CINVESTAV-IPN
Biblioteca de Ingeniería Eléctrica

FB0000009651

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS

INSTITUTO POLITECNICO NACIONAL

DEPARTAMENTO DE INGENIERIA ELECTRICA

SECCION DE COMPUTACION

PL/M-86 CONCURRENTE

Tesis que presenta el Lic. José Oscar Olmedo Aguirre para obtener el grado de MAESTRO EN CIENCIAS en la especialidad de INGENIERIA ELECTRICA.

Trabajo dirigido por el Dr. Manuel E. Guzmán Rentería

México D.F. Junio de 1988

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

KM
83.7
Б-10992
31-VII-84

DEDICADO A MI FAMILIA

MI MAMA MARIA DE LOS ANGELES AGUIRRE BARRIOS

MI TIA GLORIA AGUIRRE BARRIOS

Y MI HERMANO JORGE ALBERTO OLMEDO AGUIRRE

EN MEMORIA DE

MI ABUELA MARIA EDUVIGES BARRIOS

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

CONTENIDO

PREFACIO 1

INDICE DE CONTRIBUCIONES 3

CAPITULO I: PROGRAMACION CONCURRENTE 4
- LOS ELEMENTOS DEL LENGUAJE 6
  - MODULARIZACION 6
  - SINCRONIZACION 7

CAPITULO II: EL LENGUAJE PL/M-86 8

CAPITULO III: EL NUCLEO DEL SISTEMA OPERATIVO XINU 14
- ADMINISTRADOR DE MEMORIA 16
- ADMINISTRADOR DE PROCESOS 22
  - LISTAS DE PROCESOS 23
  - CAMBIO DE CONTEXTO 24
  - OPERACIONES SOBRE PROCESOS 27
- CALIFICADORES 29
  - EL CALIFICADOR DOPROCESS 36
- SINCRONIZACION Y COMUNICACION 40
  - SEMAFOROS 40
  - COMUNICACION DIRECTA 43
  - COMUNICACION INDIRECTA: PUERTOS 44
  - MENSAJES COMPARTIDOS 47
- CLASES Y MONITORES 55
- INTERRUPCIONES 62
  - ADMINISTRADOR DEL RELOJ 63
- INICIACION DEL SISTEMA 65

CAPITULO IV: APLICACIONES 67
- PROBLEMAS CLASICOS DE CONCURRENCIA 67
  - LA RELACION PRODUCTOR CONSUMIDOR USANDO CLASES . . . 67
  - LA RELACION PRODUCTOR CONSUMIDOR USANDO MENSAJES . . 71
  - LOS FILOSOFOS 72
  - LOS LECTORES Y ESCRITORES 74
  - LOS FUMADORES 77
- PASCAL CONCURRENTE 80

CONCLUSIONES 87

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

BIBLIOGRAFIA 88

APENDICE I: EXTENSIONES AL LENGUAJE 91
TIPOS 92
OPERACIONES 93
SENTENCIAS 105

APENDICE II: LISTADOS 108
DEFINICIONES Y VARIABLES GLOBALES: CONF.DEC 109
LISTAS DE PROCESOS: Q 111
ADMINISTRADOR DE MEMORIA: MEM 115
ADMINISTRADOR DE PROCESOS: PROC 119
SEMAFOROS Y COMUNICACION DIRECTA: SEM 126
ADMINISTRADOR DE RELOJ: CLOCK 131
COMUNICACION POR PUERTOS: PORTS 135
CLASES Y OBJETOS: DATA 141
INICIACION: SYSINIT 146
ENTRADA Y SALIDA: ES 149
CLASES Y MONITORES: CLASS 154
OPERACIONES EN XINU: XINU.EXT 154

APENDICE III: MENSAJES DE ERROR 155

PREFACIO

En la programación concurrente se incluyen las técnicas de programación para tratar problemas que involucran múltiples actividades que se desarrollan con un paralelismo potencial.

Aun cuando tiene sus origenes en el estudio de los sistemas operativos es necesario distinguirlos. La tendencia es aumentar el grado de abstracción de estos conceptos hasta convertirlos en construcciones de un lenguaje de programación.

En la actualidad existen varios lenguajes de programación para la escritura de programas concurrentes en ambientes centralizados: Pascal Concurrente [Brinch Hansen, 1975], Modula [Wirth, 1977], Pascal-Plus [Welsh y Bustard, 1979] son extensiones al Pascal secuencial y CSP/k [Holt, 1978] está basado en PL/I. Otros lenguajes de alto nivel se han propuesto para ambientes distribuidos: Communicating Sequential Process [Hoare, 1978] y Ada [1980] entre los mas conocidos, se distinguen por las facilidades que ofrecen para la comunicación entre procesos.

El propósito de esta tesis es extender el lenguaje de desarrollo PL/M-86 para facilitar la programación concurrente en un ambiente centralizado. El resultado es el lenguaje PL/M-86 CONCURRENTE.

La ampliación del lenguaje consiste en definir un conjunto de estructuras de datos, procedimientos y construcciones en los propios términos del lenguaje de desarrollo. A este conjunto se le llama núcleo. El núcleo fué tomado del sistema operativo XINU [Colmer, 1981] por su claridad y sencillez de diseño.

El material está organizada de la siguiente forma. El capitulo I es una introducción a la programación concurrente donde se introducen algunos de los términos usados en este trabajo. El capitulo II describe las caracteristicas del lenguaje de desarrollo. El capitulo III es la

descripción del núcleo de XINU. Incluye los temas de administración de bajo nivel de memoria, procesos, sincronización y comunicación. Se describen construcciones adicionales como las clases y los monitores. El manejo de interrupciones se muestra con un administrador de reloj en tiempo real. El capítulo IV ilustra el uso del lenguaje con algunos problemas clásicos de concurrencia. Por último, se describe cómo llevar este trabajo para extender otro lenguaje de programación como Pascal, ejemplificándolo con la construcción de rutinas de servicio de interrupción.

Esta tesis no es solo el trabajo de implantación del núcleo de concurrencia. Presenta también conceptos y técnicas originales que nacieron con este trabajo. Se encuentran a lo largo de la exposición y sólo aquellos que considero más importantes se les hace referencia en un índice especial de contribuciones.

PL/M-86 CONCURRENTE tiene aplicaciones en las áreas de lenguajes de programación, sistemas operativos, comunicaciones, simulación e ingeniería de software. La programación concurrente es un tema importante de investigación actual.

Para comprender mejor este trabajo, el lector debe poseer conocimientos de sistemas operativos y estar familiarizado con la arquitectura del procesador 8086/8088, así como de un lenguaje estructurado como PL/I, Pascal o C.

Deseo agradecer al Dr. Manuel Guzmán Rentería el interés que despertó en mí por la programación concurrente, y su apoyo para la realización de este trabajo. Al Dr. Renato Barrera y al M. en C. César Galindo por la cuidadosa lectura y acertadas observaciones que realizaron.

A mi amigo, M. en C. Feliú Sagols no sólo por la revisión de este material, sino también, por su valioso ejemplo durante los estudios de maestría. A mi hermano, Ing. Jorge Olmedo por la lectura y sugerencias hechas a este trabajo.

INDICE DE CONTRIBUCIONES

Las aportaciones o puntos de mayor interés de este trabajo son los siguientes:

1. El núcleo de concurrencia de XINU está escrito completamente en el lenguaje PL/M-86. No existe ninguna parte producida en un lenguaje de bajo nivel.

2. La estructura del descriptor de procesos se ha modificado de manera que no se requiere guardar todos los registros del procesador; solo se conservan dos registros que permiten guardar y restaurar el ambiente del proceso. En consecuencia, se propone un método para realizar el cambio de contexto [pp. 24-27].

3. Se describe una técnica muy versátil para obtener la información de un procedimiento a tiempo de ejecución como la dirección del código, sus parámetros actuales, etc. Esta técnica, que llamo "calificadores", permite extender el lenguaje agregando nuevas sentencias [pp. 29-36].

4. Se describe una nueva forma de crear y destruir procesos [pp. 36-39].

5. Posee la capacidad de encapsular los datos y sus operaciones mediante clases (tipos abstractos de datos) [pp. 55-60].

6. Proporciona una construcción muy frecuente en la solución de problemas de concurrencia: el monitor [pp. 60-61].

7. La comunicación indirecta entre procesos usa mensajes cuyo tipo es definido por el programador [pp. 44-46].

8. Provee un método muy poderoso para la comunicación entre procesos por medio de los mensajes compartidos. Como una ilustración de este método, presento una solución al problema clásico de concurrencia debido a Patil [1971]: los fumadores [pp. 47-54].

9. Las ideas de este trabajo se pueden llevar a cualquier otro lenguaje de programación. La sentencia adicional IOBEGIN/IOUNTIL, convierte a un procedimiento en una rutina de servicio de interrupción en el lenguaje Pascal [pp. 80-86].

CAPITULO I

PROGRAMACION CONCURRENTE

En un proyecto, como la construcción de un edificio, el trabajo se divide en etapas que realizan varios departamentos desarrollando diferentes actividades. Cada departamento puede hacer su propio trabajo con cierta independencia. Sin embargo, las actividades del proyecto que se realizan en forma paralela deben estar sincronizadas. Usamos el término concurrente para denotar un paralelismo potencial en el desarrollo de las actividades de un sistema de producción.

Esta forma de trabajo nos parece natural porque los problemas de concurrencia se presentan en la mayoría de las actividades humanas cuando se involucran a varias personas, departamentos o máquinas, los cuales poseen una autonomía local pero que están sujetos a las condiciones generales que establece el proyecto.

De este ejemplo, haremos dos observaciones importantes. Primero, si hay varios trabajadores disponibles para incorporarse al proyecto, entonces pueden trabajar simultaneamente en sus propias actividades. No obstante, el proyecto no establece esto como una necesidad sino como una consideración para mejorar el rendimiento global, es decir, un mayor número de trabajadores logrará una mayor productividad. La segunda observación se refiere a una separación del trabajo que apunta hacia una mayor especialización tanto de las actividades como de los trabajadores.

En los sistema de cómputo hay razones similares. La eficiencia consiste en el uso mas productivo de los recursos del equipo, lo que se refleja en una respuesta mas rápida a las exigencias del usuario. La abstracción permite tratar los problemas a un nivel conceptual para concentrarse en la estructura de la organización y cuál será su imagen externa o la forma de tratar con él.

El lenguaje de programación representa este grado de abstracción. La programación concurrente consiste de un conjunto de técnicas para escribir

programas que involucran a múltiples actividades que pueden realizarse en paralelo.

A continuación daremos algunos de los usos mas frecuentes de la programación concurrente: la ejecución en paralelo, simulación, control de dispositivos externos y sistemas operativos.

Ejecución en paralelo. Los grupos de sentencias, así como las expresiones, pueden tener diferentes partes que se ejecuten concurrentemente. Por ejemplo, para hacer una búsqueda en un arreglo o en un archivo, a un sistema con dos procesadores le tomaría en promedio la mitad del tiempo que le llevaría al sistema con un procesador, porque las comparaciones entre los diferentes elementos del arreglo son independientes. Para programas que requieran de un gran esfuerzo computacional, el tiempo de ejecución de un programa puede reducirse substancialmente.

Con el advenimiento de los microprocesadores resulta cada vez más atractivo el dividir los programas en varias tareas concurrentes. En el futuro encontraremos que las computadoras están compuestas por una colección de pequeños elementos de procesamiento de uso general o especializado. De hecho, la computadora IBM PC/XT para la cual está escrito este trabajo, consta de un microprocesador 8088 de propósito general y un coprocesador numérico 8087.

Simulación. Algunos programas simulan actividades que ocurren simultaneamente. Una forma de realizar esta simulación consiste en representar cada actividad mediante un programa independiente. En un juego de video, por ejemplo, los adversarios, el medio ambiente, etc., se pueden representar por programas concurrentes que poseen un comportamiento propio que evoluciona con el juego.

Control de dispositivos externos. Las computadoras de propósito específico se usan para controlar dispositivos como sensores y voltímetros ya que están asignadas a supervisar procesos industriales, experimentos, etc. Una forma de tratar estos problemas es asociar un programa a cada dispositivo externo. Tal programa es responsable de enviar y recibir las señales del medio que supervisa, realizar cálculos y producir reportes de las actividades externas.

Sistemas operativos. Los sistemas operativos constituyen el ejemplo más notable de la programación concurrente. Consiste de una colección de módulos de programas y estructuras de datos. Esos módulos reciben las solicitudes de los usuarios (p.ej. correr un programa) y administran los recursos del sistema (procesador, memoria, etc.) para satisfacer dichas necesidades. Aun cuando la programación concurrente tiene sus orígenes en los sistemas operativos es necesario distinguirlos.

La tendencia es escribir los sistemas operativos en un lenguaje de alto nivel para el desarrollo de sistemas como el PL/M-86 o lenguaje

ensamblador. Sin embargo, dichos lenguajes no proveen de los elementos básicos para escribir programas concurrentes.

LOS ELEMENTOS DEL LENGUAJE

Hay ventajas para usar lenguajes de alto nivel en vez de lenguaje ensamblador. Los programas en lenguaje de alto nivel son más fáciles de entender, probar, modificar y transportar de una computadora a otra. Con lo cual, el tiempo para el desarrollo y prueba del sistema es menor.

La generación de código ineficiente ya no es una desventaja en los lenguajes de alto nivel. Esto se resuelve aplicando varias técnicas de optimización de código. En particular, el lenguaje PL/M-86 posee hasta cuatro niveles de optimización de código, cuyo resultado es comparable al producido por un buen programador.

Los mecanismos que debe proporcionar el lenguaje para facilitar la escritura de programas concurrentes son: modularización y sincronización [Peterson y Silverschatz 1981].

MODULARIZACION

Hay tres tipos de módulos que debe proveer el lenguaje de alto nivel: procesos, procedimientos y clases (tipos abstractos de datos).

Procedimientos. Son unidades pasivas del programa que realizan una secuencia de operaciones sobre los datos. Son las estructuras elementales para el ocultamiento de la información bajo el nombre del procedimiento.

Procesos. Son módulos secuenciales activos que corren asíncronamente. Son las piezas fundamentales para la construcción de programas concurrentes.

Clases (tipos abstractos de datos). Proveen un mecanismo para encapsular variables y procedimientos, de acuerdo a las propiedades de los datos. Se requiere de cierta flexibilidad en los mecanismos de declaración de estructuras para crear clases de objetos que no habían sido anticipadas en el diseño del lenguaje y que el programador puede llegar al formular.

SINCRONIZACION

El lenguaje debe proveer protección contra los errores dependientes del tiempo, que resultan al actuar varias entidades independientes. Algunas construcciones que proporciona el lenguaje son:

Semáforos. Es un mecanismo no estructurado usado para sincronizar procesos y asegurar el acceso exclusivo de los datos.

Regiones críticas. Es un tipo especial de clase que provee el acceso seguro y eficiente a los datos.

Monitores. Proporciona mecanismos de sincronización para compartir los datos de la clase entre varios procesos.

Mensajes. Es un medio de comunicación y sincronización muy poderoso usado también en ambientes de procesamiento distribuido.

Antes de ver como estos elementos fueron incorporados al lenguaje, se dará una revisión general del lenguaje de desarrollo.

CAPITULO II

EL LENGUAJE PL/M-86

PL/M-86 es un lenguaje de alto nivel diseñado para accesar las caracterìsticas del procesador 8086/8088 para un empleo òptimo de los recursos del sistema. Haremos una breve descripciòn del lenguaje que nos servirà de referencia para los capìtulos posteriores. El lector interesado en obtener mas informaciòn puede consultar la Guìa del Usuario PL/M-86 [Intel 1980].

PL/M-86 es un lenguaje estructurado por bloques: cualquier sentencia en un programa es parte de un bloque. Un bloque es un grupo de sentencias que comienzan con la palabra clave DO o una declaraciòn de procedimiento y terminan con la palabra END. Los bloques se pueden anidar y etiquetar.

Se pueden declarar cuatro tipos de objetos por medio de un nombre simbòlico: etiquetas, constantes, variables y procedimientos. Solo una declaraciòn debe existir por cada nombre usado en el bloque. Esta declaraciòn debe aparecer al principio del bloque o bien en un bloque màs externo. Solo despuès de haberse declarado y definido los objetos se pueden usar. La definiciòn de las etiquetas, constantes y variables es su uso en el programa. Para los procedimientos, el conjunto de sentencias comprendidas en su bloque constituyen su definiciòn.

La sentencia no ejecutable DECLARE introduce alguno de los tres tipos de objetos: etiqueta, constante o variable. Los nombres simbòlicos siguen las reglas usuales de los identificadores, una letra seguida de letras o dìgitos o subrayas.

Una etiqueta es la ubicaciòn de una instrucciòn ejecutable. La palabra LABEL declara un nombre como etiqueta.

Las constantes pueden ser una substituciòn literal de texto a tiempo de compilaciòn, o bien, datos sin cambio a tiempo de ejecuciòn. La palabra clave LITERALLY define una macro substituciòn sin paràmetros. La palabra clave DATA permite usar el nombre simbòlico de un dato por su valor.

Hay siete tipos elementales de datos identificados por las palabras clave: BYTE, WORD, DWORD, INTEGER, POINTER, SELECTOR y REAL. PL/M-86 es un lenguaje consistente en el manejo de tipos: las variables, expresiones y procedimientos que devuelven un valor son de algún tipo.

Un valor de tipo BYTE es una cantidad de 8 bits. El tipo WORD se representa por dos bytes contiguos. El tipo DWORD ocupa 4 bytes contiguos de memoria. Estos tres tipos son considerados como enteros sin signo.

Un valor de tipo INTEGER representa a un entero con signo y ocupa dos bytes consecutivos. La aritmética de punto flotante es posible con el tipo REAL con formato corto, que emplea cuatro bytes, uno para el exponente y tres para la mantisa. El tipo apuntador POINTER puede contener una dirección completa de memoria formada por una porción base selector y por un desplazamiento, ambos de dos bytes de largo. El tipo SELECTOR constituye la porción base de el tipo POINTER. Una variable solo puede pertenecer a alguno de estos siete tipos.

Las variables basadas se usan para accesar indirectamente al objeto cuya direccion es apuntada por una variable que le sirve de base. La palabra BASED debe aparecer a continuación de la variable que se declara, y después, la variable que le sirve de base. Este tipo no crea espacio de memoria, solo informa al compilador que debe generar el código apropiado para hacer referencia indirecta y se usa en las operaciones con apuntadores, por ejemplo, en programas de manejo dinámico de memoria o pasaje de parámetros por referencia.

Un arreglo es una lista contigua de elementos del mismo tipo referidos por un subíndice. El primer elemento de la lista es el que posee índice cero. El número de elementos del arreglo se indica en la declaración.

La palabra clave STRUCTURE, especifica una estructura, que es una colección de elementos de diferente tipo, referenciados a través del identificador del componente.

La sentencia PROCEDURE declara una secuencia de instrucciones como un procedimiento. Los parámetros formales se declaran a continuación del identificador del procedimiento encerrados entre paréntesis. No se permiten variables basadas, subindices, o componentes de estructuras en la lista de parámetros formales. Las funciones son procedimientos que pueden devolver un valor que corresponde a uno de los tipos elementales.

Una expresión es de algún tipo elemental y puede ser una constante, variable simple, componente de una estructura, elemento de un arreglo o el valor que devuelve una función. Las expresiones aritméticas pueden formarse por medio de los operadores usuales: "+", "-", "*", "/" y MOD. Las expresiones lógicas son de tipo BYTE, obtenidas a partir de los operadores relacionales: "<", ">", "=", "<=", ">=" y "<>"; y de los operadores lógicos: AND, OR, XOR y NOT.

Las expresiones de tipo apuntador se pueden obtener a partir del operador "@" que devuelve la dirección de una variable o procedimiento y por la función BUILDPTR que obtiene una dirección completa a partir de la base de tipo SELECTOR y del desplazamiento de tipo WORD. Las expresiones de tipo SELECTOR se obtienen de la función SELECTOROF que devuelve la parte base de un apuntador. La porción desplazamiento de un apuntador se obtiene de la función OFFSETOF que es de tipo WORD.

La representación del tipo POINTER depende del modelo de segmentación de memoria elegido para compilar el programa.

El signo "=" se usa en el asignamiento de expresiones a una variable. Es posible hacer asignamientos embebidos dentro de una expresión, en cuyo caso, se debe usar el signo ":=".

Una sentencia puede ser:

La sentencia vacia: ";".
La sentencia de asignamiento.
La sentencia bloque: DO; sentencias END;
IF expresión lógica THEN sentencia
IF expresión lógica THEN sentencia ELSE sentencia
DO variable = inicial TO final; sentencias END;
DO variable = inicial TO final BY paso; sentencias END;
DO WHILE expresión logica; sentencias END;
DO CASE expresión; sentencias END;
CALL procedimiento;
CALL procedimiento(parametros actuales);
RETURN;
RETURN expresión;
GOTO etiqueta;

Cualquier sentencia debe terminar con punto y coma. La sentecia DO CASE selecciona solo una de las proposiciones de su bloque de acuerdo con un valor entero no negativo que evalúa la expresión. La invocación de un procedimiento se consigue con la sentencia CALL. Un procedimiento puede terminar al concluir las sentencias que lo definen o, explícitamente, por la sentencia RETURN. En el caso de funciones, el valor que determina es la expresión que aparece en la sentencia RETURN, que es obligatoria.

Los procedimientos pueden poseer los atributos PUBLIC, EXTERNAL, REENTRANT o INTERRUPT escritos al final de la declaración del encabezado. Un atributo informa al compilador de las características del procedimiento que debe tener en cuenta para la generación del código.

Los atributos PUBLIC y EXTERNAL extienden el alcance de los objetos. El atributo PUBLIC declara que la definición de un objéto se extiende fuera de su módulo. EXTERNAL declara el uso de un objeto definido en otro módulo. Los objetos pueden ser variables o procedimientos.

El atributo INTERRUPT permite declarar que un procedimiento responderá a alguna condición externa como la interrupción producida por un dispositivo. El procedimiento de servicio no debe tener parámetros, ni debe ser función.

El atributo REENTRANT permite suspender temporalmente la ejecución de un procedimiento. Los procedimientos directa o indirectamente recursivos deben poseer este atributo. Las variables locales se alojan en la pila de ejecución para obtener código puro.

La operación del compilador se puede dirigir con varios controles para especificar opciones como la eficiencia del código, salida del módulo objeto, modelo de segmentación de memoria, etc. Esta línea de control debe comenzar con el signo "$" en la primera columna.

El lenguaje PL/M-86 no proporciona operaciones de entrada y salida, por lo que se definieron procedimientos básicos para usarse en este trabajo. Las operaciones de entrada y salida son dirigidas a la consola de la PC. Las operaciones de lectura son hechas del teclado y las de salida escriben directamente sobre la memoria de video monocromo. La lista se encuentra en el archivo ES.P86 del apéndice II y contiene procedimientos para leer y escribir bytes, cadenas de caracteres y palabras en hexadecimal, etc.

El siguiente texto es un programa escrito en el lenguaje PL/M-86.

```
 1   $large code

 2   Arreglos: do;
 3   $include (common.lit)
 4   $include (io.ext)

 5       writechr: procedure (b) external;
 6           declare b byte;
 7       end writechr;

 6       declare n literally '100';
 7       declare vector (n) byte, x word;

 8       leearreglo: procedure (p) word;
 9           declare p pointer;
10           declare v based p (*) byte, b byte, x word;

11           x = 0;
12           do while (x < n - 1) and ((b := getchar) <> CR);
13               if b <> ' ' then
14                   do;
15                       v (x) = b;
16                       x = x + 1;
```

```
17                    end;
18              end;
19              v (x) = 0;
20              return x;
21        end arreglo;

22        escribearreglo: procedure (p, c);
23              declare p pointer, c word;
24              declare v based p (*) byte, x word;

25              do x = 0 to c;
26                    call writechr (v (x));
27              end;
28        end escribearreglo;

29        call iosysinit;
30        c =  leearreglo (@vector);
31        call escribearreglo (@vector, c);
32        call iosysend;
33   end Arreglos;
```

La línea 1 es la línea de control que especifica las opciones de compilación. En este caso, establece el modelo largo de segmentación y que debe aparecer el código generado en ensamblador.

El bloque en el nivel mas externo le da nombre al módulo (ARREGLOS). Las líneas 2 y 3 causan que se incluya en el texto el contenido de los archivos COMMON.LIT e IO.EXT, que contienen las declaraciones de IOSYSINIT, IOSYSTERMINATE, GETCHAR y CR.

El atributo EXTERNAL del procedimiento WRITECHR indica que su definición es externa a este módulo y declara que posee un solo argumento de tipo BYTE. La línea 6 declara que N será substituida literalmente en el contexto del programa. Así, la variable VECTOR es un arreglo de 100 bytes.

La función LEEARREGLO es un procedimiento de tipo WORD que devuelve el número de caracteres leídos del teclado diferentes de blanco, que terminen con el fin de línea CR pero que no excedan a N. Posee un solo argumento P de tipo POINTER que indica la dirección del arreglo donde se depositarán los caracteres leídos. La variable V es un arreglo cuya base coincide con la apuntada por P, por lo que se puede manejar como el arreglo apropiado. En esta declaración, que aparece en la línea 10, se debe observar que a continuación de V sigue la palabra BASED y después la variable base P. Como V es un arreglo, antes del tipo aparece el dimensionamiento mudo "(*)".

La sentencia de repetición DO WHILE abarca las líneas 12 a 18. En la línea 12 se realiza un asignamiento embebido en el criterio de iteración. Los paréntesis forzan a que se realize este asignamiento antes de la comparación con CR.

La sentencia IF selecciona los caracteres diferentes de blanco. Se aplica sobre una sentencia bloque no etiquetada que incluye la sentencia de acceso al arreglo basado.

Las sentencias 19 y 20 ponen al cero como un terminador y regresa la cuenta de caracteres con RETURN.

El procedimiento ESCRIBEARREGLO manda a la pantalla una secuencia de C bytes de un arreglo apuntado por P. Tiene declaraciones de variables basadas similares al anterior procedimiento. El ciclo de repetición lo realiza la sentencia DO similar al FORTRAN.

El cuerpo del programa principal comprende las líneas 29 a 32. IOSYSINIT e IOSYSTERMINATE inician y terminan las operaciones de entrada y salida, respectivamente.

La línea 30 es la invocación de LEEARREGLO. El valor que devuelve se asigna a la variable global C. Su argumento es la dirección del arreglo VECTOR obtenida por la función "@". Puesto que se escogió el modelo largo de segmentación, la dirección de VECTOR se forma de una parte selector y un desplazamiento. Todas las operaciones hechas por LEEARREGLO son aplicadas a VECTOR.

Luego de haber leído la secuencia de caracteres, ESCRIBEARREGLO los escribe en la pantalla.

CAPITULO III

EL NUCLEO DEL SISTEMA OPERATIVO XINU

El diseño del sistema operativo XINU [Comer 1984] es sencillo y elegante. Se divide en ocho componentes principales, organizados en una jerarquía por capas. Los ocho grupos mencionados son:

1 Administrador de Memoria

2 Administrador de Procesos

3 Sincronización de Procesos

4 Comunicación entre Procesos

5 Administrador del Reloj

6 Administradores de Dispositivos

7 Comunicación en Redes de Computadoras

8 Sistema de Archivos

Al examinar esta lista reconocemos que la computadora ocupa la jerarquía más baja, la capa cero. Los programas de aplicación se encuentran en el nivel más alto de esta organización. Las operaciones de un nivel solo usan las operaciones del mismo nivel o de los inferiores.

El núcleo de concurrencia implantado abarca los cinco primeros niveles, del administrador de memoria al administrador del reloj.

En el centro del núcleo se encuentra el administrador de memoria que realiza las gestiones de bajo nivel para asignar y liberar la memoria del sistema. La siguiente capa es la responsable de administrar el procesador entre los procesos que son elegibles para usarlo.

El siguiente nivel provee la cordinación entre procesos por medio de semáforos. La comunicación entre procesos provee varias operaciones para el intercambio de mensajes. En el nivel más externo de esta implantación, el administrador del reloj permite retardar la ejecución de un proceso por un determinado tiempo.

La forma de interactuar con el núcleo es invocando a un conjunto de procedimientos o "servicios del sistema".

En la invocación de un servicio, el control no regresa hasta que el procedimiento no se haya completado. Si un servicio causa que un proceso se suspenda, entonces el proceso se congela en el punto de la invocación. Cuando el administrador lo activa, el control aparece en el mismo punto.

Un procedimiento consiste en el código ejecutado por un proceso. En contraste, los procesos no poseen un código exclusivo: varios procesos pueden ejecutar el mismo código simultaneamente.

Las variables, los argumentos actuales y la pila de ejecución son exclusivos de cada proceso.

La presentación de los cinco componentes de XINU, sigue el mismo orden en este trabajo. Los temas que proponen una técnica o aplicación se intercalaron en esta secuencia, como la técnica de "calificadores" y los tipos abstractos de datos.

Empezaremos con el administrador de memoria de bajo nivel.

ADMINISTRADOR DE MEMORIA

La memoria principal es un recurso esencial para la ejecución de un programa. El administrador provee dos funciones básicas: la asignación de memoria y su liberación. Para asignar memoria se debe especificar el tamaño del bloque. El administrador obtiene un bloque de memoria de un espacio reservado para este fin, llamada el area del sistema, y devuelve la dirección del bloque al proceso. Cuando se libera memoria, el proceso debe proporcionar la dirección del bloque, y entonces, el administrador la regresa al area del sistema.

Aun cuando el administrador original de XINU está diseñado para un direccionamiento lineal de memoria, se puede adaptar al esquema de memoria segmentada del 8088. Antes de ver el diseño del manejo de memoria, revisaremos la forma como organiza la memoria el compilador PL/M-86.

La memoria está organizada en cuatro segmentos fijados en la fase de encadenamiento del programa PL/M-86: la sección de código, de constantes, de datos, y de memoria.

Sección de código. Esta sección contiene el código objeto generado por el programa fuente. El compilador genera un prólogo y un epílogo para el programa principal. El prólogo inicia los registros del procesador y habilita el sistema de interrupciones.

Sección de constantes. Esta sección contiene todas las variables iniciadas con el atributo DATA, constantes reales, y listas de constantes. En el modelo largo de segmentación, esta sección esta contenida en la sección de código.

Sección de datos. Asigna espacio para todas las variables excepto parámetros, variables basadas y las ubicadas por el atributo AT, o variables locales en procedimientos con atributo REENTRANT.

Sección de la pila de ejecución. Se usa en la ejecución de procedimientos y como almacenamiento temporal. El tamaño de la pila la determina el compilador.

Sección de memoria. Esta es un area de memoria referenciada por el arreglo predefinido MEMORY. El compilador genera una sección de memoria de longitud cero, debiendo ser modificada en el encadenamiento del programa.

Todos los programas de este trabajo son compilados con el modelo de segmentación LARGE. En este modelo, se usa un segmento separado para cada sección de código de los módulos del programa, asi como, para cada sección de datos. Hay solo un segmento para la sección de la pila de ejecución, y otro segmento para la sección de memoria.

Durante la ejecución del progama, las secciones de código y datos cambian al activar procedimientos externos. Esto significa actualizar los registros CS y DS.

La dirección de la sección de la pila de ejecución se mantiene en el registro SS.

Las anteriores caracteristicas del manejo de memoria del compilador tienen dos consecuencias importantes:

1. La memoria asignada a las secciones de código y datos no las controla el administrador. Esto mantiene la integridad del código del programa y los datos.

2. El area del sistema puede residir en alguna de las secciones de datos, o en la sección de memoria.

Para ubicar el area de memoria en la sección de datos, es suficiente con declarar un arreglo suficientemente grande. Para ubicarla en la sección de memoria debe usarse el arreglo predefinido MEMORY y fijar su extensión de acuerdo a la capacidad de memoria disponible.

Ubicar el area del sistema en el arreglo MEMORY tiene la ventaja de no restringir su tamaño. Sin embargo, en este trabajo el area se ubicó en un arreglo de 8K bytes de extensión. La razón es ilustrar que el area del sistema puede ubicarse en cualquier lugar de la memoria, dependiendo de las caracteristicas de la aplicación. El administrador sólo requiere conocer la dirección del area y su tamaño.

El area del sistema se maneja como una lista de bloques ordenada por sus direcciones crecientes. Cada bloque en la lista, contiene un apuntador a la dirección del siguiente bloque de memoria y el tamaño del bloque actual.

Con esta organización para los bloques, es de esperar que el tamaño de memoria que se puede gestionar es de al menos el espacio suficiente para

guardar el apuntador y un contador. Esto asegura que cuando el bloque sea liberado, tenga la extensión suficiente para describir su propia longitud y la dirección del bloque siguiente cuando se reintegre a la lista.

Dichas consideraciones son importantes en el diseño, porque determinan las características del administrador: el tamaño del area de memoria del sistema, el tamaño máximo y mínimo del bloque que se puedé obtener y los problemas de fragmentación de memoria que acompañan a la segmentación. Al solicitar memoria de menor tamaño, no sería posible guardar la descripción del bloque y habría bloques de memoria que al no ser restaurados a la lista, no serían utilizados mas. Por otra parte, cuando se requieren bloques de 1 byte, se despediciarían varios bytes más, dependiendo del tamaño del descriptor.

La solución más sencilla que se propone, es la siguiente: el apuntador al siguiente elemento en la lista está formado solo por su parte selector o base y el contador es de una palabra de extensión.

De lo anterior, las características del administrador son las siguientes:

1. El area de memoria del sistema donde se harán las gestiones es de hasta 1 megabyte de tamaño.

2. La dirección del bloque obtenido está normalizada, es decir, la parte desplazamiento del apuntador es cero.

3. Se puede solicitar cualquier cantidad de memoria de hasta 64K bytes de extensión.

4. Aún cuando el tamaño del bloque puede ser como se indica en el primer punto, la cantidad real obtenida se redondea al siguiente parágrafo. Si se solicita un byte de memoria, el sistema devuelve 16 bytes y cuando se requieren 30 bytes, se obtienen 32 bytes.

Con esto, cualquier bloque que sea liberado se puede reasignar a la lista para usarse otra vez, aunque posiblemente se desperdicie espacio internamente.

La figura 1 representa una porción de la lista de bloques. La dirección del bloque 1 se representa por SEGB1:0000, donde SEGB1 es la parte selector de la dirección, diferente de cero. En esta dirección se guarda el segmento del bloque 2, SEGB2. Se puede determinar la dirección de la base del bloque ya que siempre tiene desplazamiento cero.

La siguiente palabra (dirección SEGB1:0002) contiene la longitud, en bytes, del bloque actual. Las demás palabras no contienen información significativa para el administrador. Los demás bloques tienen una estructura similar. El último bloque de la lista apunta al segmento cero de la memoria.

AREA DEL SISTEMA

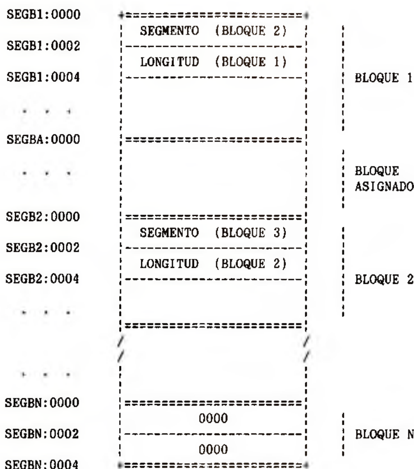


Figura 1. Lista de bloques de memoria.

Se deben hacer algunas observaciones respecto de la ubicación del area del sistema.

Es irrelevante de dónde se obtendrá la memoria desde el punto de vista del proceso, a condición de que sea un espacio contiguo cuya dirección y tamaño sean conocidos. Pero para el sistema, la situación puede ser diferente, porque puede destinar a distintas aplicaciones su propio banco de memoria. Esto se acomoda con el hecho de que el 8088 no posee mecanismos de protección de memoria. Así, un banco de memoria de 64K puede asignarse a

un programa para que no exceda esas fronteras en su uso, en el caso extremo de conservar la integridad de los demás procesos. Es particularmente útil, cuando se alojan tablas importantes del sistema y el espacio para las variables persistentes.

Por ello, convienen varios bancos de memoria de tamaño ajustable, que una sola area del sistema. El esfuerzo adicional que se introduce no es excesivo. Solo se requiere especificar la dirección del banco en las operaciones de gestión. Un procedimiento prepara el banco para usarlo, escribiendo las estructuras de lista sobre él, reservando el primer parágrafo. Allí se indica su extensión, la dirección del siguiente banco, así como la ubicación del primer bloque disponible, si lo hay.

Tal vez la aplicación más interesante de esto es en la programación persistente, cuyos objetos (datos y procedimientos) se conservan en forma independiente de los programas. Por ejemplo, las tablas importantes del sistema se obtienen de un area especial. Esta area puede usarse para intercambiar datos con memoria secundaria a intervalos regulares de tiempo o por algún evento.

Otra consideración es la seguridad. Por ejemplo, si el núcleo de XINU manejara listas ligadas para representar las colas de espera de procesos, sería preferible que esta memoria no se ubicara en el area donde se obtienen las pilas de procesos.

En este trabajo, existe solamente un area de memoria del sistema.

El procedimiento MEMINIT, prepara la estructura de bloques y el area del sistema, a partir de su ubicación y extensión. Se invoca en la iniciación del núcleo. En ese momento, hay un solo bloque que coincide con el área del sistema.

Hay dos "tipos de memoria" que un programa puede demandar: la memoria para la pila de ejecución (STACK), y la memoria del montón (HEAP). En el diseño original, la diferencia entre ellas no es solo por su uso, sino por la forma como se obtiene. La memoria para la pila se obtiene de las localidades altas del area del sistema y la memoria del montón de las localidades más bajas. La razón es la de ofrecer cierta seguridad al area del montón de las posibles colisiones con la pila de ejecución. Puesto que no hay una solución satisfactoria a este problema, en este trabajo la única distinción entre ambos tipos de memoria es por su uso.

La función GETSTACK es la función básica de gestión de memoria: asigna un bloque del tamaño requerido en bytes. La búsqueda del bloque se realiza a través de la lista con el método "First-Fit", que determina el primer bloque que tenga una extensión mayor o igual al solicitado. Si un bloque así se encuentra, se toman dos acciones:

1. Si el area del bloque en la lista es mayor que el requerido, se forma un nuevo bloque con el espacio sobrante.

2. Se extrae el bloque y se ajusta el apuntador de su predecesor.

Recíprocamente, el procedimiento FREESTACK libera la memoria asignada. Su argumento es un apuntador al extremo final del bloque. Esta información es suficiente para restaurarlo a la lista de bloques. Hay tres situaciones posibles:

1. Cuando el bloque se traslapa con otro en la lista, se produce un error.

2. Cuando colinda con otro en la lista, se produce un solo bloque de la unión de ambos, cuya extensión es igual a la suma de los dos.

3. En otro caso, el bloque se inserta en la lista.

La función GETMEM asigna un bloque de memoria del montón. La cantidad requerida se indica en bytes. Hace una llamada a GETSTACK y normaliza el apuntador que obtiene, con lo cual determina la dirección del principio del bloque.

El procedimiento FREEMEM libera la memoria obtenida por GETMEM. Sus argumentos son el apuntador al principio de bloque y su longitud en bytes.

La declaración de la estructura de los bloques de memoria aparece en el listado del archivo MEM.DEC en el apéndice II. Las operaciones MEMINIT, GETSTACK, FREESTACK, GETMEM y FREEMEM pertenecen al módulo MEM del archivo MEM.P86. El area de memoria del sistema se encuentra en la sección de datos de este módulo.

Las declaraciones de los objetos que este módulo exporta se encuentran en el archivo MEM.EXT.

ADMINISTRACION DE PROCESOS

Un proceso se representa por una estructura de datos que contiene la información que lo caracteriza como su identificador, estado, dirección de código, etc. El sistema mantiene la información de todos los procesos en la tabla PROCTAB. Cada proceso es referenciado por un identificador, que corresponde a un índice de la tabla. La estructura se muestra en el archivo PROC.DEC del apéndice II. El módulo PROC del archivo PROC.P86 define las operaciones del administrador de procesos.

El administrador asigna un identificador a cada proceso que corresponde a un índice de la tabla PROCTAB. Los identificadores válidos están comprendidos entre 0 y NPROC - 1, donde NPROC es el número máximo de procesos que puede manejar el sistema.

La diferencia principal con el diseño original de XINU de la estructura de PROCTAB es que sólo contiene los registros del procesador SS y SP que señalan la ubicación de la pila de ejecución del proceso.

En cualquier momento, un proceso puede encontrarse en uno de seis estados: en ejecución, elegible, suspendido, esperando recibir, esperando en un semáforo y dormido. Cuando una entrada de la tabla no es usada por ningún proceso, se marca como libre (PRFREE). El proceso que recibe el servicio del procesador, es el que se encuentra en estado de ejecución (PRCURR), y su identificador de proceso está indicado por la variable CURRPID. Los procesos elegibles son los que esperan el uso del procesador (PRREADY).

El algoritmo "Round Robin" se usa para seleccionar el siguiente proceso a ejecutarse. El proceso elegible con la más alta prioridad es el que se selecciona primero. Los procesos con igual prioridad esperan en una lista de elegibles, para atender primero a los que llegaron primero. Cuando está en ejecución el proceso, su identificador se remueve de la lista de elegibles. El cambio de contexto tiene lugar solo cuando el

siguiente proceso elegible tiene mayor prioridad que el que se encuentra en ejecución.

Para simplificar el algoritmo, existe siempre al menos un proceso, el proceso nulo que, o está en ejecución, o está en la lista de elegibles. Este proceso posee la prioridad más baja y consiste de un ciclo que se repite mientras exista al menos un proceso en ejecución, elegible o dormido.

LISTAS DE PROCESOS

En el núcleo de XINU se manejan varias colas para representar los estado del proceso. Comparten una estructura similar: son listas doblemente ligadas con un campo clave. Puesto que un proceso no puede tener dos estados diferentes a la vez, solo puede pertenecer a una lista. En vez de crear varias listas con igual estructura pero donde cambia dinámicamente el número de sus elementos, se ha preferido usar una sola y varios apuntadores a los primeros y últimos elementos de cada lista lógica. Esta estructura está representada por el arreglo QENT.

El número máximo de elementos es NQENT y corresponde a: el número de procesos que se puede manejar NPROC, el doble del número de semáforos NSEM que soporta, dos para la lista de procesos preparados RDYHEAD y RDYTAIL, y uno para la lista de procesos "dormidos" SLEEPQ.

Cada elemento de QENT se compone de una campo QKEY (usado especialmente en cada estado del proceso), una referencia hacia el elemento predecesor QPREV y otra hacia el sucesor QNEXT. Cada lista posee apuntadores hacia el primer y último elementos.

Las operaciones manejan colas de procesos ordenados por una clave. INSERT inserta un proceso de acuerdo con el orden creciente de su prioridad. ENQUEUE inserta un elemento al final de la lista. INSERTD inserta un proceso en la lista de procesos dormidos.

La eliminación de un elemento de la lista se realiza por DEQUEUE. GETFIRST extrae el primer elemento y GETLAST el último.

NEWQUEUE obtiene una nueva lista devolviendo el índice de la cabeza de lista. QINIT inicia las estructuras. Las demás operaciones del módulo QUEUE aparecen en el archivo Q.P86 del apéndice II.

CAMBIO DE CONTEXTO

El cambio de contexto constituye el punto central de un ambiente de multiprogramación. Consiste en detener el proceso en ejecución y guardar su estado. Entonces, el administrador elige otro proceso, restaura su ambiente y le cede el control.

El ambiente del proceso se caracteriza por el contenido de los registros del procesador y por el area de memoria del proceso, que incluye el código, el area de datos y la pila de ejecución.

Lo anterior significa, que para el correcto funcionamiento del sistema, el ambiente debe permanecer inalterado hasta que el proceso recupere de nuevo el servicio del CPU. Desafortunadamente, el 8088 no posee mecanismos de protección de memoria, de manera que no es posible garantizar en forma segura y eficiente que el área de memoria del proceso permanezca inalterada. Supondremos, entonces, que los procesos no intentarán exceder las fronteras de su memoria.

A continuación presento una modificación a la rutinas de cambio de contexto del diseño original del XINU. En esta versión aprovecho las características de generación de código del compilador PL/M-86.

Se acostumbra que el descriptor de procesos reserve un espacio para guardar el contenido de todos los registros del procesador en el momento del cambio de contexto. En este trabajo sólo se mantiene un apuntador a la dirección de memoria en el cual se guardaron los registros.

Un procedimiento en PL/M-86 que posee el atributo INTERRUPT, causa que el compilador genere código en el prólogo del procedimiento para guardar el contenido de los registros del CPU en su pila de ejecución. En forma similar, el código que genera en el epílogo restaura los registros, precisamente como ocurre en una interrupción.

Por otra parte, el compilador permite accesar los registros de la pila de ejecución, el segmento de la pila SS y apuntador al tope de la pila SP, como si se tratara de variables globales de tipo WORD. De esta forma, el apuntador al area de memoria donde se conserva el contenido de los registros se forma con los registros de la pila de ejecución SS y SP. La variable STACKBASE representa al registro segmento de la pila SS y STACKPTR al apuntador al tope de la pila SP.

```
!     !          !                       !     !          !
!     !    FL    !                       !     !    FL    !
!     !    CS    !                       !     !    CS    !
!     !    IP    !                       !     !    IP    ! <- SS:SP
!     !    AX    !                       !     !    AX    !
!     !    CX    !                       !     !    CX    !
!     !    DX    !                       !     !    DX    !
!     !    SI    !                       !     !    BX    !
!     !    DI    !                       !     !    SI    !
!     !    ES    !                       !     !    DI    !
!     !    DS    !                       !     !    DS    !
!     !    BP    ! <- SS:SP              !     !    BP    !
v     !          !                       !     !          !
```

a) Al término del prólogo b) Al término del epílogo

Fig. 1. Actividad en la pila de ejecución durante una interrupción.

El cambio de contexto se realiza por una rutina de servicio a una interrupción por "software", CTXSW, que es un procedimiento sin parámetros con el atributo INTERRUPT e invocado por la sentencia CAUSEINTERRUPT. La rutina CTXSW tiene asignado el vector 0FFH. El procedimiento SETINTERRUPT asocia la rutina de servicio con el vector de interrupción.

Se puede describir el cambio de contexto en los cinco pasos siguientes:

1. Al cabo del prólogo de CTXSW, el contenido de los registros del 8088 están guardados en la pila de ejecución. El proceso en ejecución preserva el estado del procesamiento en su pila.

2. Los registros de la pila del 8088, SS y SP, se guardan en ese momento en el descriptor del proceso aún en ejecución.

3. Se determina el nuevo proceso a ejecutar, obtenido de una lista de procesos elegibles con el algoritmo "Round-Robin".

4. Los registros del procesador, SS y SP, se reemplazan con los valores del proceso seleccionado, recuperados de su descriptor. Esto hace que la pila de ejecución cambie a la del proceso activado.

5. Al cabo del epílogo de CTXSW, el contenido de los registros se recuperan. El proceso seleccionado recupera el ambiente del procesamiento de su pila de ejecución.

Pasos 1 y 2 de CTXSW

```
                    PROCESO  A:

 |            |     FL    |
 |            |     CS    |
 |            |     IP    |
 |            |     AX    |
 |            |     CX    |
 |            |     DX    |
 |            |     SI    |
 |            |     DI    |
 |            |     ES    |
 |            |     DS    |
 v            |     BP    |     <-    SS:SP

        PROCTAB(A).SS = STACKBASE
        PROCTAB(A).SP = STACKPTR
```

Pasos 4 y 5 de CTXSW

```
                    PROCESO  B:

        STACKBASE = PROCTAB(B).SS
        STACKPTR  = PROCTAB(B).SP

 -            |     FL    |
 |            |     CS    |
 |            |     IP    |     <-    SS:SP
 |            |     AX    |
 |            |     CX    |
 |            |     DX    |
 |            |     BX    |
 |            |     SI    |
 |            |     DI    |
 |            |     ES    |
 |            |     DS    |
 |            |     BP    |
```

Fig. 2. Cambio de contexto entre dos procesos A y B.

Debe notarse que en el paso 3, si se selecciona de nuevo al proceso, el efecto de CTXSW sería el mismo que el producido por una rutina de interrupción. Esto se debe a la simetría del código producido para guardar

y recuperar el contenido de los registros.

La desventaja de este método consiste en que se basa en la suposición de que el área de memoria del proceso, que incluye a la pila de ejecución, permanezca intacta hasta que el proceso se vuelva a activar. Sin embargo, sin un esquema de protección de memoria provista por el "hardware", ningún método para conservar el estado del procesamiento es seguro, ya que algún proceso podría alterar el area de datos del sistema o de otro proceso.

Una de las ventajas consiste en que está escrito completamente en el lenguaje PL/M-86. Esto significa no sólo un aumento en la claridad de la implantación. Establece que un procedimiento de tipo especial INTERRUPT, así como la habilidad para accesar los registros de la pila de ejecución, son construcciones básicas para el control del procesador.

Con estas capacidades no sólo es posible instalar con gran facilidad rutinas de servicio de interrupción. Se puede escribir un núcleo de concurrencia breve y sencillo. Además, se pueden diseñar trazadores y depuradores de una amplia gama de sofisticación .

Otra ventaja es la eficiencia. El código generado es reducido, eficiente y rápido en ejecutarse porque no requiere mover bloques, ni calcular direcciones complicadas de memoria. Tales consideraciones son importantes en sistemas que realizan una gran cantidad de cambios de contexto donde se produce una degradación del rendimiento.

OPERACIONES SOBRE PROCESOS

Un proceso en estado suspendido no es elegible para el uso del procesador, por lo que es necesario distinguirlo de los estados en ejecución y elegible. El procedimiento SUSPEND, se aplica solo sobre procesos en ejecución o elegibles. Si el proceso es elegible, entonces se extrae de la lista de espera. Si se suspende así mismo, se ejecutará el siguiente proceso elegible.

La operación RESUME se aplica solo a procesos suspendidos haciéndolos elegibles para el uso del procesador.

KILL detiene un proceso inmediatamente y lo elimina del sistema por completo. Una llamada a FREESTACK libera la memoria que el proceso había usado como pila de ejecución. Las demás acciones que tome KILL dependerán del estado en el que se encuentre el proceso. Si el proceso se encontraba en ejecución, libera su entrada de la tabla de procesos y después cede el control a otro proceso elegible. Si estaba esperando sobre un semáforo, se incrementa el contador de éste. Finalmente, cuando el proceso es elegible

para recibir el CPU, se extrae de la lista de elegibles.

Figura 3. Diagrama de transición de estados

El procedimiento CREATE crea un nuevo proceso. La idea es obtener la imagen exacta del proceso justo como si hubiera sido interrumpido mientras corría, de tal manera que CTXSW pueda regresar mas tarde a él. CREATE encuentra una entrada libre en la tabla de procesos, cuyo índice corresponderá con el identificador de proceso. Obtiene un espacio de memoria para la pila de ejecución del tamaño solicitado. En el momento de la creación, el proceso se encuentra en estado suspendido. Siendo el procedimiento la construcción que va a representar el código del proceso, se debe indicar a CREATE, la dirección del código, la de sus argumentos, así como su número. Se indica también la prioridad con la cual el proceso va a correr en el sistema y un nombre que será asociado con el proceso. Los valores de los registros que corresponden a la imagen inicial del proceso se deben depositar sobre su pila, para que un cambio de contexto (una llamada a CTXSW) lo ponga en ejecución.

CALIFICADORES

Hay dos objetivos diferentes que se deben enfatizar en un lenguaje de programación: su eficiencia de ejecución y su expresividad. Por eficiencia nos referimos solamente a la rapidez con la que realiza las acciones la computadora. En cierto sentido, los lenguajes mas eficientes son los que realizan exactamente las acciones que se indican, es decir, el lenguaje de máquina. La expresividad se refiere a la facilidad con la cual se puede describir un comportamiento y probar que es correcto. Un lenguaje es más expresivo cuanto más pueda emplearse para describir relaciones complejas en los términos más simples.

Un lenguaje que maximize la eficiencia no necesariamente es el que tiene mejor rendimiento. Simplemente, porque el programador puede tardar más tiempo en especificar como expresarlo en vez de qué expresar. El objetivo de introducir un nuevo lenguaje de programación ha sido, generalmente, el de conseguir una mayor expresividad por encima de las consideraciones de eficiencia. Esta tendencia se debe a la aparición de procesadores más rápidos y poderosos a un costo menor.

En esta sección, se desarrolla una técnica que permitirá definir varias construcciones de alto nivel que darán al lenguaje PL/M-86 CONCURRENTE una mayor expresividad y abstracción. La expresividad se deriva de las varias acciones que establece el lenguaje que se desarrollen sin que el programador tenga conocimiento directo de ellas.

Un CALIFICADOR es un procedimiento que obtiene y/o modifica la información contenida en la pila de ejecución. Esta definición es muy importante en este trabajo.

La versatilidad de las técnicas basadas en calificadores quedará demostrada en las construcciones que se presentan: las clases, los monitores, la creación y destrucción de procesos, y la instalación de rutinas de servicio de interrupción.

Por ejemplo, en el lenguaje PL/M-86 CONCURRENTE, el procedimiento es

la unidad que representa al proceso. Cuando se crea, el administrador de procesos necesita conocer la información del procedimiento como la dirección del código, el tamaño de la pila de ejecución, sus argumentos actuales, etc. El siguiente texto declara al procedimiento PRODUCER, e ilustra la creación, activación y destrucción de una instancia del proceso. Como se aprecia, no hay una forma sintáctica que especifique al proceso ni las instancias de él.

```
producer: procedure (x);
     declare x word;
     declare i word;

     do i = 0 to x;
          call writehex (i, ' ');
     end;
     call kill (getpid);
end producer;
```

la sentencia KILL(GETPID) de PRODUCER es la autodestrucción del proceso.

La siguiente sentencia crea y activa una instancia con identificador PID:

```
pid = resume (create (@producer, 256, 16, @('PROCA', 0), 2, @(20, 0)));
```

donde @PRODUCER es la dirección del procedimiento, con una pila de ejecución de 256 bytes de tamaño y con prioridad 16. El identificador "PROCA" se usa con propósitos de depuración. El argumento 20 de PRODUCER, de 2 bytes de extensión, es el número de veces que se repetirá el ciclo de escritura.

Con la técnica que se propone se define la sentencia estructurada DO PROCESS que califica al procedimiento como un proceso. Cuando el control alcanza la sentencia, se crea y activa el proceso. Cuando la abandona, el proceso se destruye.

```
producer: procedure (x);
     declare x word;
     declare i word;

     do process;
          do i = 0 to x;
               call writehex (i, ' ');
          end;
     endp;
end producer;
```

para crear una instancia del proceso basta con invocar al procedimiento:

```
call producer (20);
```

Ambos programas producen el mismo resultado, pero es evidente la superioridad expresiva del segundo. La sentencia DO PROCESS no solo oculta varios detalles sino que es más fácil de aprender y usar. Esta sentencia oculta por macroprocesamiento, la llamada del calificador DOPROCESS, determinando la información necesaria a partir del contexto del programa donde se invoque.

Para saber cómo obtiene esta información, veamos que ocurre, en general, en una secuencia de llamada a un procedimiento.

En una secuencia de llamada, los parámetros actuales (si los hay) son puestos en la pila de ejecución antes de ejecutar la instrucción CALL del 8088. Puesto que el apuntador al tope de la pila avanza hacia abajo cuando se inserta un dato, el primer parámetro ocupa la posición más alta y el último parámetro ocupa la mas baja.

Después de pasar los parámetros, la instrucción CALL coloca la dirección de regreso en la pila. En el caso "LARGE", el tipo de regreso depende de que el procedimiento sea local o público. Si es local, la dirección de regreso es un desplazamiento relativo al segmento de código de 2 bytes de extensión. Si es público, la dirección se compone de una parte segmento y otra desplazamiento, cada una de dos bytes.

A continuación se ejecuta el prólogo, que es un fragmento de código que el compilador inserta al principio del procedimiento, cuyo propósito es el de preservar el ambiente de la ejecución previo a la llamada. El prólogo hace lo siguiente:

1. Cuando el procedimiento posee el atributo PUBLIC y el programa tiene el modelo de segmentación largo, el contenido del registro DS se pone en la pila. El registro DS se actualiza con un valor que se encuentra en el segmento de código y que representa el area de datos del procedimiento.

2. El apuntador a la base BP se guarda en la pila y se actualiza con el valor del registro SP. Esto permite direccionar correctamente los parámetros actuales del procedimiento y, si posee la opción REENTRANT, podrá accesar correctamente las variable locales alojadas en la pila.

3. El control pasa a las sentencias ejecutables del cuerpo del procedimiento.

```
PROLOGO:
          PUSH DS
          PUSH BP
          MOV  BP, SP

CODIGO:
          . . .
```

EPILOGO:

```
        MOV SP, BP
        POP BP
        POP DS
        RET <OFS>
```

Figura 1. Prólogo y epílogo de un procedimiento para el modelo largo de segmentación.

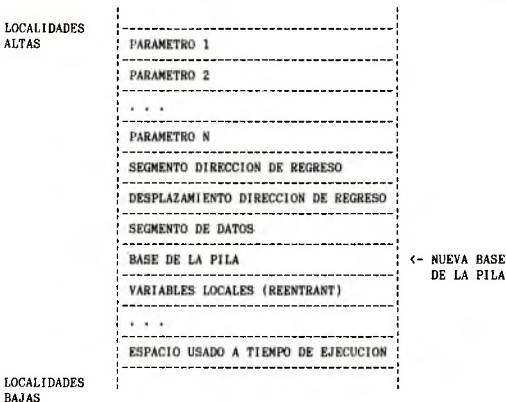


Figura 2. Estado de la pila de ejecución al término del prólogo.

En forma similar, el epílogo es una secuencia de código que el compilador inserta al final de procedimiento. Las acciones del epílogo son las siguientes:

1. Si se reservó espacio para el almacenamiento temporal y variables locales durante la ejecución, el registro SP se actualiza con el valor del registro BP, para liberar este espacio.

2. El valor previo de los registro BP y DS se recuperan de la pila de ejecución.

3. La dirección de regreso se restaura de la pila y se libera el espacio asignado a los parámetros actuales del procedimiento.

Al término de la ejecución del prólogo, la pila contiene una descripción parcial del procedimiento que incluye:

a. La dirección de la instrucción siguiente a la invocación.

b. El valor de los parámetros actuales.

c. El segmento de datos del procedimiento previo.

Parece adecuado, que luego de ejecutarse el prólogo del procedimiento, se invoque un procedimiento calificador, que sea la primera sentencia ejecutable, para que recupere esta información. El lenguaje PL/M-86 provee un acceso directo a los registros segmento de la pila y apuntador al tope de la pila del procesador 8088, por medio de las variables de tipo WORD, STACKBASE y STACKPTR, respectivamente. Con ambas variables, es posible accesar el contenido de la pila interpretándola como un arreglo de tipo WORD de 32K elementos cuya base está en la dirección STACKBASE:0000.

El índice de un elemento de este arreglo se obtiene dividiendo por 2 el desplazamiento de la localidad de memoria correspondiente, que se encuentra en el rango de 0 a 64K. El lector debe tener en cuenta que las direcciones contenidas en la pila se refieren a direcciones de bytes, pero que las operaciones CALL, RET, PUSH y POP del 8088 empilan o desempilan palabras porque es el tamaño de los registro IP, SP, BP, etc.

El siguiente módulo define el calificador DOINSPECTION. Determina la información asociada con el procedimiento que lo invoque.

```
$large
calificador: do;

    declare (CS, IP, IP1, DS, SS, SP, BP0, BP1, BP2, NAR, NVL) word public;

    doinspection: procedure public;
        declare stackp pointer, stack based stackp (*) word;

        SS  = stackbase;
        SP  = stackptr;
        stackp = buildptr (sel (SS), 0);
        BP0 = shr (SP, 1);
        BP1 = shr (stack (BP0), 1);
        BP2 = shr (stack (BP1), 1);
        DS  = stack (BP0 + 1);
        CS  = stack (BP0 + 2);
        IP  = stack (BP0 + 3);
        IP1 = stack (BP1 + 2);
```

```
          NAR = shr(BP2 - BP1, 1) - 4;
          NVL = shr(BP1 - BP0, 1) - 8;
     end doinspection;

     . . .
end calificador;
```

En el siguiente programa, el procedimiento PRO es examinado por el calificador DOINSPECTION.

```
$large
calificadores: do;

     doinspection: procedure external;
     end doinspection;

     pro: procedure (a, b);
          declare a byte, b integer;

          call doinspection;
          . . .
     end pro;

     . . .
     call pro (1, 2);
     . . .
end calificadores;
```

El significado de las variables es el siguiente:

SP es el desplazamiento del tope de la pila.

SS es el segmento de la pila.

BP0 es el índice del arreglo asociado con el ambiente de DOINSPECTION.

BP1 es el índice asociado con el ambiente de PRO.

BP2 está relacionado con el ambiente del programa principal.

DS es el segmento de datos del módulo CALIFICADORES.

CS es el segmento de código del mismo módulo.

IP es la dirección de regreso de DOINSPECTION. Coincide con la dirección del cuerpo del procedimiento calificado PRO.

IP1 es la dirección de regreso de PRO en el programa principal.

NAR es la cantidad en bytes que ocupan los argumentos actuales.

NVL es espacio en bytes reservado para las variables locales (si las hay).

BP1 + 3 es el índice de la última palabra del último argumento actual de PRO.

BP2 - 1 es el índice de la primera palabra del primer argumento actual de PRO.

Esta técnica permite recuperar la historia de las activaciones de los procedimientos, retrocediendo con el apuntador a la base de la pila BP. Esta descripción es bastante completa y la obtiene del contexto del programa donde se haga la invocación del calificador.

Esta es una nueva forma de extender un lenguaje. Un calificador puede considerarse como una sentencia adicional del lenguaje, que especifica que el procedimiento al que pertenece en su definición, es un tipo especial de construcción. Se puede considerar como una alternativa a la bien conocida técnica de preprocesamiento, aunque no son exclusivas. Puesto que el tipo de sentencias que se agregan con los calificadores al lenguaje son más bien declarativas, parece ser menos general, aunque esto ya depende del tipo de aplicación. En el lenguaje PL/M-86 CONCURRENTE, el preprocesamiento se combina perfectamente con esta técnica ya que oculta la invocación del calificador mediante un juego de palabras similares a una sentencia, para dar uniformidad y consistencia a la notación.

Las desventajas de los calificadores pueden provenir de la aplicación que se les dé o de la forma como fué realizada, ya que puede usarse hasta para reproducir ciclos de repetición que es, evidentemente, un uso nada conveniente. Por ejemplo, podemos agregar al lenguaje PL/M-86 la sentencia de iteración REPEAT UNTIL. Con esta sentencia, podemos hacer una búsqueda secuencial en el arreglo A del valor V:

```
i = 0;
repeat;
   i = i + 1;
until ((i < 100) and (a(i) = v));
```

el calificador inicial REPEAT obtiene la dirección de la siguiente sentencia I = I + 1;, usada por el calificador final UNTIL para trasladar allí el control, hasta que la condición en su argumento sea verdadera.

Se debe resaltar que esta técnica se aplica más bien en declaraciones que son interpretadas a tiempo de ejecución, en vez de hacerlo en una fase previa a la compilación. Esto tiene dos consecuencias importantes. Primero, los componentes de un programa declarados con calificadores (instancias de clases, procesos, etc.) son creados y destruidos dinámicamente. Segundo, se pueden definir nuevos tipos de objetos que son operados sólo por un conjunto específico de procedimientos y, por ello, requieren ser

calificados. Este enfoque modular es consistente con la programación estructurada.

La tendencia de los lenguajes actuales es el de proporcionar un mayor control de los recursos del sistema, por ello, esta técnica no se restringe al lenguaje PL/M-86. Para diseñar un calificador se requiere conocer la arquitectura del procesador y el lenguaje de desarrollo. Se necesita conocer los modos de direccionamiento de memoria, el juego de instrucciones y el uso de los registros del procesador. Del lenguaje de desarrollo, se debe saber si permite el acceso directo o indirecto de los registros del procesador, el código que genera, los modelos de segmentación que ofrece, etc. Por ello, extender un lenguaje con calificadores es más difícil, pero tiene la ventaja de que concentra la dificultad en torno al diseño del sistema y no al del trabajo del programador.

Las extensiones del lenguaje basadas en calificadores resultan en una disminución de la frecuencia de errores en la escritura de programas. Su uso es más parecido al de una sentencia que al de un procedimiento, porque depende del contexto del programa donde se encuentre. Para usar correctamente un calificador se deben especificar un conjunto de reglas sintácticas en vez de formular los argumentos que requiere, en número y tipo.

EL CALIFICADOR DOPROCESS

El calificador DOPROCESS establece una forma sencilla y elegante de crear procesos sin recurrir a CREATE, por lo que no se requiere conocer la dirección del código, el tamaño de la pila de ejecución, etc. Cualquier procedimiento calificado de esta forma, en el momento de su invocación será preparado para ejecutarse concurrentemente. Así, para el lenguaje PL/M-86 CONCURRENTE, invocar el procedimiento calificado con DOPROCESS corresponde a crear y activar el proceso.

Para crear un proceso se requiere información acerca del procedimiento que lo va a representar. La idea consiste en definir un calificador que explore la información contenida en la pila de ejecución en el momento de la invocación del procedimiento, y luego, forzar a que regrese al programa principal.

Para aclarar esto, examinemos el siguiente fragmento de programa:

```
producer: procedure;
     doprocess;
     do loop;
```

```
                call writestr (@ ('PRODUCE', 0), cr);
            end;
        end producer;

        consumer: procedure;
            doprocess;
            do loop;
                call writestr (@ ('CONSUME', 0), cr);
            end;
        end consumer;

        . . .
        call writestr (@ ('ANTES',0), cr);
        call producer;
        call consumer;
        call writestr (@ ('DESPUES',0), cr);
        . . .
```

El calificador DOPROCESS debe estar diseñado para que obtenga la información requerida y produzca el siguiente resultado, al correr el programa:

```
ANTES
DESPUES
```

en lugar de:

```
ANTES
PRODUCE
PRODUCE
PRODUCE
. . .
```

Se necesita que DOPROCESS produzca este resultado para dar la impresión de una ejecución concurrente, ya que PRODUCER contiene un ciclo infinito en su definición que impediría invocar los procedimientos que le siguen después y, por lo tanto, crear los procesos.

El cambio de contexto requiere que el estado de la pila sea el mismo que el producido por una interrupción. Esto sugiere que DOPROCESS sea un procedimiento sin parámetros con el atributo INTERRUPT al igual que CTXSW, la rutina de cambio de contexto. Podemos suponer que al invocar a PRODUCER el proceso ya fué creado e interrumpido por vez primera por DOPROCESS. Entonces, la imagen de la pila en este momento debe ser similar a la que tome CTXSW en el cambio de contexto.

Las imagenes de las pilas solamente son "similares" porque cada proceso tiene su propia pila con ubicación y tamaños diferentes, originando pequeños cambios en los contenidos. Por ejemplo, aunque los registros AX o CS no cambien al cambiar la pila, los registros BP, SP y SS si lo hacen,

debiendo ser actualizados.

Al conocer las direcciones donde están guardados los contenidos de los registros, el calificador también puede actualizar el contenido del registro IP de modo que regrese directamente al programa principal y no a la siguiente sentencia del procedimiento.

Con la descripción completa del procedimiento, DOPROCESS puede crear el proceso y ponerlo en la lista de elegibles para uso del CPU, esperando el primer cambio de contexto que dará el sistema cuando termine de examinar los demás procedimientos, dando la clara ilusión de concurrencia.

Las acciones que toma el calificador DOPROCESS se pueden resumir en:

1. Determinar si es posible crear otro proceso. De no ser así terminar la ejecución en una condición de error.

2. Determinar la posición de la pila de ejecución para obtener: registros, variables locales, argumentos actuales, etc.

3. Crear un bloque de memoria de tamaño fijado en STACKSIZE bytes. Este bloque será la pila del proceso.

4. Copiar el contenido de la pila de ejecución en la pila del proceso.

5. Actualizar el registro en la pila del proceso BP.

6. Asignar la dirección de la pila del proceso a las variables SS y SP del descriptor.

7. Insertar el identificador del proceso en la lista de elegibles para el CPU y ponerlo en estado "READY". Asignarle la prioridad INITPRIO.

8. Obtener la dirección de la siguiente sentencia a la invocación del procedimiento en el programa principal y regresar.

Los resultados que producen DOPROCESS y CREATE son compatibles, excepto en la forma como obtienen la información.

La ventaja de CREATE sobre DOPROCESS es que el programador puede establecer el tamaño de la pila de ejecución, así como la prioridad inicial del proceso.

La capacidad de elegir el tamaño de la pila puede ser un factor importante dependiendo del tipo de procesos que se maneje. Para la mayoría de los problemas, sólo se requiere que los procesos tengan un espacio en la pila adecuado para la evaluación de expresiones, cambio de contexto o interrupciones. Sin embargo, los procesos que declaren muchas variables locales referidas a la pila de ejecución, requieren de un gran espacio.

SINCRONIZACION Y COMUNICACION

Una tècnica de bajo nivel de coordinaciòn entre procecesos es a travès de la sentencia RESCHEDULE o de los procedimientos SUSPEND y RESUME. Cuando varios procesos pueden entrar a una regiòn crìtica es posible garantizar que solo uno entre, suspendiendo a los demàs. Al abandonar la regiòn crìtica, se reanuda la ejecuciòn de los otros procesos que deseen accesar dicha regiòn.

La desventaja de tales tècnicas es que se requiere conocer informaciòn adicional al problema que se resuelve. Se necesita saber, por ejemplo, cuàles son los procesos que accesan la regiòn crìtica, causando una pèrdida en la independencia de los procesos.

Las primitivas de sincronizaciòn de alto nivel son los semàforos, comunicaciòn directa e indirecta.

SEMAFOROS

Un semàforo es un objeto compuesto por una variable de estado del semàforo, un contador y apuntadores al primer y ùltimo elementos de la lista de procesos en espera. El conjunto de semàforos està representado por una tabla. Cada semàforo està identificado por un ìndice de la tabla.

La estructura de los semàforos y sus operaciones se declaran en el mòdulo SEM en el archivo SEM.P86.

Las operaciones definidas sobre los semàforos son SCREATE, SDELETE, WAIT, SIGNAL, SCOUNT, SRESET y SIGNALN.

Un proceso puede crear un nùmero arbitario de semàforos siempre que no

Aunque esto no es frecuente en PL/M-86 porque las variables son referidas al segmento de datos y no a pila de ejecución, a menos que el procedimiento posea el atributo REENTRANT, discutimos la posibilidad en general. Este problema se puede resolver haciendo de STACKSIZE una variable que sea fijada al valor deseado antes de la invocación del procedimiento, o bien, modificar a DOPROCESS para que obtenga esta información como un parámetro del procedimiento.

La prioridad del proceso no representa dificultad ya que puede cambiarse usando los servicios del sistema. Al iniciar los procesos conviene que tengan la misma prioridad.

Crear un proceso con este método tiene la ventaja de que se pueden pasar los parámetros actuales sin importar su número o tipo. Esto se puede usar para comunicarse con el programa principal o proceso creador en la iniciación de variables.

Un ejemplo de la creación dinámica de procesos y su comunicación por parámetros se dará en el tema de sincronización con semáforos, calculando el factorial de un entero.

El calificador DOPROCESS se define en el módulo PROC del archivo PROC.P86 del apéndice II. Se puede comparar con el texto de CREATE para comprobar que la complejidad de ambos procedimientos es similar.

exceda al tamaño de la tabla, con la función SCREATE que devuelve el identificador del semáforo. Cuando se crea, debe especificarse el valor inicial del contador que debe ser mayor o igual a cero. SDELETE libera el semáforo del proceso. Si había procesos esperando en este semáforo, el sistema los convierte en elegibles para uso del CPU.

Las demás operaciones sobre semáforos involucran el valor del contador. Este valor es devuelto sin cambio por la función SCOUNT. El procedimiento WAIT decrementa el contador. Si el contador se hace negativo, inserta al proceso en la lista de espera del semáforo y reasigna el CPU a otro proceso. De otra forma, WAIT devuelve el control al proceso.

La operación SIGNAL incrementa el contador. Si el contador es positivo, SIGNAL devuelve el control al proceso. De otra forma, toma el primer proceso de la lista de espera y lo hace elegible.

Es frecuente hacer varias llamadas a SIGNAL con un mismo semáforo. En tal caso, SIGNALN proporciona el mismo resultado pero con mayor eficiencia pues reduce el número de cambios de contexto.

La operación SRESET, cambia el valor del contador por el indicado, que debe ser no negativo. Si el contador era negativo, tiene el mismo efecto de realizar varias llamadas a SIGNAL. Si era positivo o cero, simplemente ajusta el contador a su nuevo valor.

Sin embargo, cualquiera que sea la operación que se realize, se cumple que un contador no negativo implica que su lista de espera está vacía y, por el contrario, una cuenta negativa N significa que hay N procesos esperando en el semáforo.

El siguiente programa calcula el factorial de un número y muestra como usar semáforos como primitivas de sincronización.

```
$large

factorial: do;
$include (xinu.ext)

     declare s semaphore, d word;

     factorial: procedure (prev, n, p) reentrant;
          declare prev semaphore, n word, p pointer;
          declare next semaphore, d based p word, f word;

          do process;
               if n > 1 then
                    do;
                         next = screate (0);
                         call factorial (next, n - 1, @f);
                         call wait (next);
```

```
                    d = f * n;
                    call sdelete (next);
               end;
          else
               d = 1;
          call signal (prev);
     endp;
end factorial;

do concurrent;
     s = screate (0);
     call factorial (s, 7, @d);
     call wait (s);
     call sdelete (s);
     call writestr (@ ('Factorial (7) =',0), ' ');
     call writehex (d, cr);
endc;

end factorial;
```

Este programa produce:

Factorial (7) = 13B0

que es el valor correcto del factorial de 7 escrito en hexadecimal (no hay un procedimiento que escriba en decimal).

El factorial de un entero N se obtiene creando otro proceso que calcule el factorial de N - 1, siempre que N > 1. Si N <= 1, devuelve 1.

Para poder calcular el factorial de N se deben crear N instancias de FACTORIAL. El proceso que evalúa la función para el valor de N debe crear otro que calcule la función para N - 1. Para sincronizarse, crea un semáforo NEXT (lo inicia con 0) y espera en él. Cuando el otro proceso obtiene el valor de la función para N - 1, señala a NEXT indicando que ya puede proseguir. Entonces, termina el cálculo del factorial, indicando a su vez al proceso que lo creó que ya puede continuar.

Este programa es muy interesante por varias razones. Demuestra la capacidad del lenguaje para crear y destruir dinámicamente, instancias de procesos y semáforos.

Un punto que debe resaltarse, es la comunicación entre los procesos por medio de sus parámetros. Aun cuando en un parámetro el paso es por referencia, no se debe pensar que es posible remplazar esta comunicación por el modelo de memoria compartida. Cada proceso posee una área de datos propia. La comunicación por parámetros en este ejemplo establece un flujo de datos entre esas áreas: el flujo de valores que se deben evaluar entre el proceso FACTORIAL de N y el de N - 1; y el flujo de resultados de

evaluaciones en el sentido opuesto.

Posiblemente este ejemplo sea demasiado sencillo para demostrar la capacidad que posee el lenguaje para distribuir la carga de trabajo entre varios procesos. Sin embargo, demuestra que es posible y señala una forma de hacerlo. Con esta capacidad se incrementa el grado de concurrencia de las aplicación que se desarrollen.

COMUNICACION DIRECTA

Es la primera de las dos formas de comunicación por pasaje de mensajes. Esta diseñada para asegurar que el proceso no se bloqueará mientras envía los mensajes y que consumirá, tan solo, una cantidad fija de memoria para su representación.

Las características de esta comunicación son:

1. Es una comunicación directa entre procesos.

2. Los puertos sólo pueden almacenar a lo más un mensaje de tipo INTEGER.

3. La comunicación sólo se puede establecer entre dos procesos.

4. Los mensajes son enviados por copia.

5. La comunicación es asimétrica. El emisor debe nombrar al receptor. El receptor sólo toma el mensaje.

Si varios mensajes se envían a un proceso sólo se recibe el primero. De esta forma, un proceso puede determinar cuál, de entre varios procesos, realizó determinado evento. Es decir, después de recibido el primero, cualquier otro mensaje será descartado por el sistema. Esto significa, también, que en esta forma de comunicación ningún proceso será bloqueado al intentar enviar un mensaje.

Un mensaje corto está implementado como dos componentes de la estructura de datos del proceso receptor: PMSG y PHASMSG. Si el proceso tiene un mensaje que ha arribado, este ha sido almacenado en PMSG y el campo PHASMSG indica que el mensaje fué recibido con un valor diferente de cero. Sin embargo, si espera un mensaje, el proceso entra en un estado especial: esperando recibir.

Tres operaciones básicas están definidas para esta clase de mensajes: SEND, RECEIVE y RECIEVECLR.

El procedimiento SEND emite un mensaje al proceso especificado. Si dicho proceso estaba esperando recibir un mensaje, se reactiva haciéndolo elegible para el uso del CPU y permitiéndole accesar dicho mensaje.

La función RECEIVE obtiene el mensaje, si éste ya habla sido enviado al proceso. Sino, el proceso se pone en espera y abandona el control. Es el proceso que le envla el mensaje quien lo reactiva.

Por último, RECEIVECLR también obtiene un mensaje cuando está disponible. De otro modo, la función regresa de inmediato con un mensaje especial.

COMUNICACION INDIRECTA: PUERTOS

Un puerto puede considerarse como un objeto donde los mensajes son depositados y tomados por los procesos. Cada puerto posee un identificador usado en las operaciones de flujo de mensajes.

Este método de comunicación posee las siguientes caracterlsticas:

1. Es una comunicación indirecta entre procesos.

2. Los puertos tienen la capacidad de almacenar un número fijo de mensajes.

3. Un proceso puede tener asociado más de un puerto.

4. Un puerto puede usarse por más de un par de procesos.

5. Los puertos proveen almacenamiento temporal automaticamente para conservar los mensajes recibidos.

6. Los mensajes son enviados por copia.

7. La comunicación es simétrica. Ambos, emisor y receptor, deben conocer el puerto. La lista de parámetros es idéntica en ambas operaciones.

8. El tipo de los mensajes es definido por el programador, por lo que son de longitud fija.

La última característica no pertenece al diseño original del sistema operativo XINU, y significa que los mensajes son de cualquier tipo, simple o estructurado, en vez del exclusivo tipo INTEGER.

El sistema provee las siguientes operaciones básicas sobre los puertos:

1. Crear un puerto.
2. Enviar y recibir mensajes a través del puerto.
3. Destruir el puerto.

Las operaciones para enviar y recibir mensajes están diseñadas de manera que si dos procesos tratan de recibir un mensaje del puerto, sólo el primero de ellos que hizo la gestión lo obtendrá.

Cada puerto consiste de una cola de mensajes de la longitud requerida; de una variable STATE, que indica el estado del puerto (libre o asignado); de dos semáforos, SSEM, que controla el número máximo de mensajes que pueden enviarse y, RSEM, que controla su disponibilidad y de dos apuntadores, TAIL y HEAD, que indican el más reciente y el más antiguo mensaje recibido, respectivamente. La cola de mensajes está implementada con un apuntador BUFR a una cola circular representada por un arreglo. Cada mensaje es un elemento del arreglo. El puerto gestiona la cantidad de memoria, AREA, necesaria para representar un arreglo de COUN mensajes de longitud SIZE. Esto se debe a que un puerto se crea durante la ejecución del programa y no sería posible reservar el espacio que requiera en el puerto, sino tan solo un apuntador a él.

El módulo PORTS contiene las declaraciones de los puertos y sus operaciones. Aparece en el archivo PORTS.P86 del apéndice II.

El conjunto de puertos del sistema está representado por una tabla de puertos PORTS. Un puerto esta identificado por un índice en esta tabla. Un proceso puede demandar cualquier número de puertos siempre que no exceda al total disponible. El sistema inicializa la tabla de puertos poniéndolos en estado libre.

La función PCREATE devuelve el identificador del puerto creado con la capacidad requerida de mensajes. Busca una entrada libre en la tabla de puertos. Si la encuentra, debe crear los semáforos SSEM y RSEM. Inicializa RSEM con cero y SSEM con la capacidad del puerto.

El procedimiento PSEND envía una copia del mensaje al puerto indicado, que se añade a la cola circular. Si ya no hay espacio disponible en el puerto, entonces el proceso espera en SSEM. PSEND, reactiva el proceso que esperaba recibir mensaje cuando hay uno disponible. Dicho proceso es obtenido de RSEM. El apuntador TAIL avanza cuando se recibe la copia del mensaje.

El procedimiento PRECEIVE obtiene un mensaje cuando hay uno

disponible. El mensaje se copia de la cola circular del puerto a la dirección de un "buffer" del proceso. Cuando no hay mensajes disponibles, el proceso espera en el semáforo RSEM. PRECEIVE hace elegibles a los procesos que esperaban en SSEM. El apuntador HEAD apunta al mensaje que va a ser enviado al proceso.

Cuando el semáforo de mensajes recibidos tiene una cuenta no negativa N, significa que hay N mensajes esperando en el puerto; una cuenta negativa -N significa que hay N procesos esperando un mensaje.

El procedimiento PDELETE libera el puerto del proceso marcándolo como libre. Se liberan al mismo tiempo los semáforos SSEM Y RSEM. Si habla procesos esperando sobre alguno de esos semáforos, se ponen en la lista de elegibles. Se devuelve al sistema el bloque de memoria asignado al puerto.

MENSAJES COMPARTIDOS

En la siguiente sección se proponen nuevas operaciones para realizar la comunicación indirecta entre procesos usando la misma estructura de mensajes.

La diferencia con las operaciones PSEND y PRECEIVE consiste en que si dos procesos intentan recibir un mensaje, ambos lo obtendrán.

Las razones para introducir estas operaciones se justifican al considerar la diversidad de problemas cuyo comportamiento se puede expresar en estos términos.

Considérese un sistema donde la toma de una decisión depende de algún evento. Un diseño altamente modular del sistema consistiría en escribir procesos especializados que representen a cada decisión. Reducir el grado de acoplamiento entre los procesos necesitaría que ellos mismos observaran el evento y se comportaran de acuerdo a las reglas que se preescriben.

Un ejemplo muy sencillo de esto es un juego de cartas. Cada jugador espera, en cierto momento, una carta que complete su juego. Las cartas que se sacan del montón se colocan sobre una mesa para que sean observadas por todos los jugadores. Esta es la idea principal. Hay reglas generales que un jugador debe cumplir, como la prioridad para tomar una carta. Sin embargo, la característica esencial del problema es la capacidad de que todos los elementos del sistema compartan de una fuente común, cierta información.

Los problemas que se derivan no sólo de trasmitir la información, sino también, de sincronizar las acciones que aseguren su consistencia, sugiere usar alguna forma de comunicación indirecta entre procesos.

Figura 1. Operaciones PSEND y PRECEIVE.

La comunicación indirecta usando las operaciones PSEND y PRECEIVE se ilustra en la Figura 1.

En esta figura, se muestra la comunicación entre un proceso emisor y tres receptores. El emisor envía al puerto tres mensajes, que consisten en los caracteres 'P', 'L' y 'M'. La numeración de los receptores sirve para indicar el orden en el que reciben los mensajes. Cada receptor extrae un mensaje que es exclusivo para él. En consecuencia, un receptor no puede esperar tener acceso a todos los mensajes que se produzcan.

La siguiente figura ilustra la situación ideal de los mensajes compartidos.

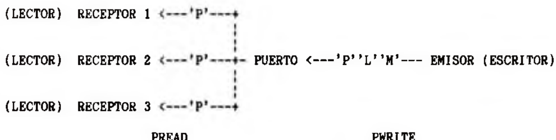


Figura 2. Operaciones PREAD y PWRITE (mensajes compartidos).

Esta figura muestra el caso donde todos los receptores comparten el mismo mensaje 'P' que es el primero que arriba al puerto. En una etapa posterior, los tres receptores obtienen el mismo mensaje 'L'.

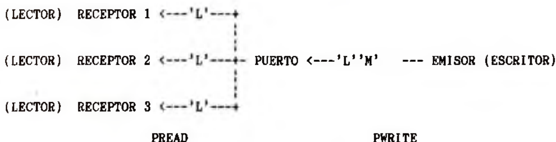


Figura 3. Los receptores comparten otro mensaje.

Aun cuando puede haber varias soluciones para este problema, la que se eligió tiene la virtud de la sencillez de la implantación. Por otra parte, los puertos y los mensajes son parcialmente compatibles con los mensajes ya estudiados. Es decir, se pueden aplicar las operaciones PCREATE, PDELETE y PCOUNT, pero no PSEND y PRECEIVE.

No se pretende resolver el problema en su generalidad ya que es muy difícil y depende del tipo de sistema que se represente, con el fin de definir criterios de optimización.

La solución que se propone define nuevas operaciones sobre los puertos, PWRITE y PREAD, para enviar o recibir mensajes que en adelante llamaré compartidos. El control del tráfico de mensajes lo hace un proceso especial, PCONTROLLER, definido por el sistema, el cual llama a PCONTROL.

Para ejemplificar el uso de los mensajes compartidos plantearemos el siguiente problema:

Se trata de descompactar y homogenizar el texto que proviene de una fuente, bajo las siguientes reglas:

A. Un dígito D, entre '0' y '9', dentro del texto, significa que el caracter siguiente debe repetirse D veces exactamente.

B. Las letras minúsculas deben convertirse a mayúsculas.

C. Cualquier otro caracter permanece sin cambio.

El siguiente programa TEXT resuelve este problema. El proceso WRITER produce una secuencia de caracteres que representa al TEXTO. Cada caracter se escribe sobre una cinta TAPE usando la operación PWRITE.

La cinta es examinada por tres procesos lectores READERA, READERB y READERC, cada uno de los cuales representa una de las reglas de traducción del texto. Los lectores reciben el mismo caracter al aplicar PREAD, y después deciden si se aplica la regla.

El proceso PCONTROLLER debe actuar explícitamente sobre el puerto que administra.

Por último, la traducción del texto es reensamblada y escrita en la pantalla.

```
$large save nolist

text: do;
$include (xinu.ext)

    declare tape port, c byte;
```

```
writer: procedure (tape) reentrant;
     declare tape port;
     declare p pointer, s based p (128) byte, i byte;
     do process;
          p = @('5!3 PL/M-86 Concurrente3 5! ', 0);
          i = 0;
          do while s (i) <> 0;
               call pwrite (tape, @s (i));
               i = i + 1;
          end;
          call writechr (cr);
     endp;
end writer;

readerA: procedure (tape) reentrant;
     declare tape port;
     declare (c, d, i) byte;
     do process;
          do loop;
               call pread (tape, @c);
               if ('0' <= c) and (c <= '9') then
                    do;
                         call pread (tape, @d);
                         do i = 1 to c - '0';
                              call writechr (d);
                         end;
                    end;
          end;
     endp;
end readerA;

readerB: procedure (tape) reentrant;
     declare tape port;
     declare c byte;
     do process;
          do loop;
               call pread (tape, @c);
               if ('a' <= c) and (c <= 'z') then
                    call writechr (c - 'a' + 'A');
          end;
     endp;
end readerB;

readerC: procedure (tape) reentrant;
     declare tape port;
     declare c byte;
     do process;
          do loop;
               call pread (tape, @c);
```

```
                if not (('a'<=c) and (c<='z') or ('0'<=c) and (c<='9'))
                    then call writechr (c);
            end;
        endp;
    end readerC;

    do concurrent;
        tape = pcreate (1, size (c));
        call writer  (tape);
        call readerA (tape);
        call readerB (tape);
        call readerC (tape);
        call pcontroller (tape);
    endc;
end text;
```

El programa produce el siguiente resultado:

```
!!!!!   PL/M-66666666 CONCURRENTE   !!!!!
```

que es correcto de acuerdo con el texto: 5!3 PL/M-86 Concurrente3 5!

A continuación trataremos la implantación de los mensajes compartidos.

En lo que sigue, habrá que distinguir dos clase de procesos involucrados en los mensajes compartidos: escritores y lectores. Un proceso se considera escritor cuando envía un mensaje al puerto, y lector, cuando lo recibe de él.

Para hacer sencillo el diseño, se impondrá la regla de que cualquier lector no puede demandar mensajes con anterioridad a su solicitud. Un lector no puede recibir mensajes retrasados. Esta restricción es razonable, puesto que así ocurre en las situaciones más comunes: cuando se escuchan las noticias de la radio, se reciben desde el momento en que se sintoniza la estación; si se pierde la señal, se pierden mensajes que no se pueden recuperar.

Las razones por las que un proceso lector pierda mensajes pueden ser que haya sido suspendido, haber estado esperando sobre un semáforo distinto de los que provee el puerto o permanecer dormido.

Con el fin de evitar atascamientos, supondremos que existe al menos un proceso escritor. Podemos considerar que los lectores y escritores constituyen un grupo de procesos, relacionados lógicamente por el uso de un puerto para el intercambio de mensajes. Un proceso pertenece al grupo por declaración explícita. Un proceso adicional, provisto por el sistema y que

corre con muy baja prioridad, administrará las operaciones sobre el puerto.

A diferencia de las anteriores operaciones con puertos, los procesos lectores no reactivan a los ecritores o viceversa. Ambos, lectores y escritores son reactivados por el administrador del puerto PCONTROLLER.

El procedimiento PWRITE permite a un proceso escritor enviar una copia del mensaje al puerto. Si el puerto aún posee capacidad, PWRITE devolverá el control al proceso. De otra forma, el proceso quedará bloqueado en el semáforo SSEM. PWRITE, actualiza el apuntador TAIL al último mensaje enviado, pero no reactiva a los procesos lectores que puedan estar esperando en RSEM.

El procedimiento PREAD obtiene una copia de un mensaje para el proceso lector que lo emite. A diferencia de PRECEIVE, antes de obtener el mensaje, PREAD bloquea siempre al proceso en el semáforo RSEM. Cualquier número de lectores puede obtener un mensaje del puerto. PREAD no reactiva a los procesos escritores que esperan sobre SSEM, ni avanza al apuntador HEAD.

El procedimiento PCONTROL administra el tráfico de mensajes. El proceso que invoca esta operación, PCONTROLLER, toma la prioridad más baja del grupo. Cuando todos los procesos del grupo han sido bloqueados en alguno de los dos semáforos SSEM o RSEM, entonces PCONTROL se pone en ejecución. Este administrador tiene por función el determinar si hay mensajes disponibles, en cuyo caso, reactiva todos los procesos lectores que se encuentran esperando en RSEM, restableciendo su contador a cero con SRESET. Puesto que él controla el avance del apuntador HEAD, todos los lectores obtendrán el mismo mensaje.

Cuando eventualmente, vuelva a reactivarse, PCONTROL avanzará a HEAD para obtener un nuevo mensaje. Este ciclo se repite hasta haber agotado todos los mensajes depositados por los escritores. Entonces, el administador entra en su segunda etapa, desbloquear a los procesos escritores que esperan en SSEM, poniendo el valor del contador a la capacidad de mensajes del puerto con SRESET. Esto completa el ciclo de lectura y escritura. PCONTROL termina su ejecución cuando no hay más mensajes producidos por los escritores. Destruye el puerto por una invocación a PDELETE y devuelve la ejecución a PCONTROLLER.

Se puede establecer un ciclo de lectura y escritura de la siguiente forma: un proceso lector se bloquea cuando emite un PREAD; un proceso escritor se bloquea cuando no hay espacio en el puerto por una invocación a PWRITE; cuando todos los procesos del grupo se han bloqueado, se ejecuta PCONTROLLER, que reactiva a los lectores; cuando todos los mensajes del puerto se hayan terminado, reactivará a los escritores, reanudándose el ciclo.

El comportamiento que se representa equivale a esperar indefinidamente a que un lector tome su mensaje, siempre y cuando esté en ejecución o sea elegible para el procesador. La ventaja de este enfoque es que pueden

aparecer nuevos lectores o desaparecer los ya existentes.

Las operaciones usuales sobre los puertos, PSEND y PRECEIVE, y las operaciones con mensajes compartidos, PWRITE y PREAD, no deben combinarse sobre un mismo puerto, pues deriva en un mal funcionamiento. Las operaciones sobre mensajes compartidos son más generales que las usuales, aunque también menos eficientes, y por ello no se necesitan.

Por otra parte, el sistema no verifica que siempre exista al menos un proceso escritor y un proceso lector para evitar atascamientos. Esta clase de problemas se presentan en general con todas las primitivas de sincronización y por ello no se intenta dar una solución a este problema, dejando al usuario la responsabilidad de asegurarse que siempre existan lectores y escritores.

Es necesario subrayar que la adminstración de mensajes se hace sólo sobre una familia de procesos. Con la implantación actual, PCONTROLLER no se activaría si hubiera siempre algún proceso de mayor prioridad que no perteneciera al grupo. Esto constituye una restricción muy severa. Para simplificar el trabajo, todos los procesos de usuario pertenecen a un solo grupo.

Una forma de resolver este problema es realizar un cambio en el diseño del núcleo. Todos los procesos pueden pertenecer a algún grupo determinado. Las prioridades de los procesos son relativas sólo al grupo al que pertenecen. Los grupos, a su vez, pueden poseer una prioridad. Los grupos son administrados por el sistema, que pueden recibir el uso del CPU por un cantidad de tiempo que depende de su prioridad. Dentro del grupo, los procesos son administrados como antes.

Para realizar esta solución, sólo se requiere agregar un componente a la estructura QENT, la lista de procesos elegibles, para que siempre queden consecutivos cuando se inserten. Con esta solución, en un sistema multiusuario, los procesos ya no se pueden adjudicar una prioridad grande para que corran siempre primero en el sistema. Este esquema difiere con el de colas multinivel ya que los procesos no se ordenan respecto a su prioridad, sino respecto a su grupo.

Otra desventaja de esta implantación de los mensajes compartidos es que aumentan considerablemente el esfuerzo en la administración de procesos, puesto que los lectores se desactivan cuando intentan leer tan solo un mensaje. Los escritores producen mensajes por periodos determinados por la capacidad del puerto. Qué ocurre cuando un nuevo escritor produce mensajes más rápidamente? Es evidente que se debe acortar el periodo de escritura. Para este problema, una solución parcial la constituye la conocida técnica de niveles de agua o "water marks". Consiste en impedir que los lectores consuman todos los mensajes del puerto hasta un cierto nivel. Entonces, el periodo de escritura se acorta, ya que no se leen todos los mensajes. Más tarde, los lectores reanudan su actividad hasta llegar al siguiente nivel repitiéndose el proceso.

Es necesario observar que el criterio de terminación del administrador de mensajes puede arrebatar, repentinamente, el uso de un puerto. Cómo saber que un proceso que no ha emitido una operación sobre el puerto no lo hará mas tarde? Lo ideal sería conservar indefinidamente al proceso administrador al igual que el proceso nulo.

Una alternativa es fijar un tiempo de tolerancia para los lectores. Al agotarse este tiempo, el administrador remueve el mensaje y permite que un escritor pueda producir otro. El tiempo es una forma de disminuir la degradación del rendimiento. El diseñador puede definir otro evento conforme a su aplicación.

Después de la lista anterior de desventajas de los mensajes compartidos, qué puede justificar su uso? La gran variedad de problemas en que se puede aplicar. Desafortunadamente, la extensión del tema limita a mencionar sólo un par de aplicaciones.

La primera aplicación esta orientada a los lenguajes de programación, que enfaticen un diseño basado en flujo de datos en vez del flujo de control. Una aplicación estaría representada por procesos concurrentes en los cuales se define una entrada y una salida de información con la cual se puede comunicar con otros procesos.

Los mensajes compartidos permiten una variedad más amplia de construcciones que las que se podrían obtener con los "tubos" que corresponden a las operaciones PSEND Y PRECEIVE. Un sistema de información desarrollado sobre un lenguaje así se vería como una red de procesos comunicados por puertos.

La otra aplicación se refiere a la forma de construir herramientas de programación como editores, compiladores e intérpretes. Al reducir el acoplamiento entre los diversos módulos de un sistema resulta fácil escribir, por ejemplo, un analizador sintáctico independientemente del intérprete o del traductor. Por su capacidad de ser dirigida por interrupciones, estas tareas se podrían ejecutar simultaneamente al proceso de edición.

El módulo PORTS se encuentra en el archivo PORTS.P86 y las declaraciones en PORTS.DEC en el apéndice II.

CLASES Y MONITORES

La idea principal de la programación con objetos es la de estructurar los datos y su acceso en el lenguaje de programación [Peterson y Silverschatz 1981]. Una característica notable del lenguaje PL/M-86 CONCURRENTE es su capacidad de tratar con objetos. La programación con ojetos continúa la linea de los lenguajes estructurados como Pascal. Una clase es una construcción del lenguaje para representar los datos junto con sus operaciones. Un objeto es una instancia de una clase.

El lenguaje PL/M-86 establece que los datos pertenecen a solo un dominio de valores o tipo, aún cuando su representación interna pueda ser idéntica. El propósito de los tipos de datos es el de usar consistentemente los operadores y funciones en la evaluación de expresiones. Cualquier intento de combinar dos expresiones de diferente tipo es rechazado por el compilador a menos de que se establezca una conversión explícita a través de funciones.

Mientras tratamos con datos de tipos elementales estos mecanismos son suficientes para asegurar un uso consistente, pero cuando elaboramos estructuras más complejas (p.ej. pilas, colas o listas ligadas) el lenguaje no proporciona mecanismos para estructurar operaciones como lo hace para estructurar datos (arreglos, estructuras, etc.).

El lenguaje SIMULA 67 [Dahl et al. 1972] introdujo el concepto de clase para declarar datos junto con el conjunto de procedimientos que constituyen las únicas operaciones válidas que se pueden aplicar. Este mecanismo provee un medio para que el diseñador estructure su programa de acuerdo con las propiedades de los datos. Un ejemplo frecuente es la cola, que consiste de un area de espera sobre la cual se pueden aplicar las operaciones de agregar elementos y extraer el primero en espera. Un intento de accesar directamente a cualquier elemento de la cola es una contradicción con su definición, por lo que una cola debe considerarse como una clase.

Se puede definir una clase en el lenguaje PL/M-86 CONCURRENTE como un conjunto de declaraciones de variables globales y procedimientos en un módulo independiente del programa principal. Los procedimientos con atributo ENTRY son los métodos de acceso a las clases y se distinguen de los procedimientos locales. Los métodos de acceso contienen sólo una de las sentencias, DO ASSIGN o DO EXCLUSION.

Una clase posee un método especial de asignación calificado por la sentencia DO ASSIGN. Debe ser el primer método que se aplique a un objeto. Este método crea dinámicamente una instancia de la clase y la asigna a un identificador de objeto, al mismo tiempo que inicia las variables globales.

Los demás métodos de acceso son calificados por la sentencia DO EXCLUSION. Recupera el area de datos asociada con el objeto, de manera que la operación se aplique directamente sobre las variables que representan el estado del objeto.

La única comunicación entre una clase y el exterior, es a través de los parámetros del método, es decir, el ámbito de las variables de la clase está restringido a su módulo. Puesto que la clase es un objeto estático (sólo contiene declaraciones) la única forma de activar a la clase es por la ejecución de un método.

Cuando se aplica un método a un objeto, se debe especificar su identificador como el último parámetro en la lista de argumentos. Esta es una forma de notación descriptiva para relacionar el objeto y su operación.

Se pueden crear objetos dinámicamente. La pertenencia de un objeto a una clase la establece el método de asignación.

La pila es una estructura de datos muy frecuente que puede representarse convenientemente con clases. Una pila con representación contigua es una estructura compuesta por un area de almacenamiento y un apuntador al último elemento insertado o primero en extraerse. El módulo STACK implanta a la pila junto con los métodos de creación, inserción y eliminación llamados CREATESTACK, PUSH y POP, respectivamente.

```
$large
stack: do;
$include (class)

     declare   n       literally  '10';
     declare   tstack  literally  'structure (t integer, a (n) integer)';
     declare   stack   tstack;

     createstack: procedure (s) entry;
          declare s object;
          do assign;
               stack.t = n;
```

```
          enda;
     end createstack;

     push: procedue (d, s) entry;
          declare d integer, s object;
          do exclusion;
               if stack.t > 0 then
                    do;
                         stack.t = stack.t - 1;
                         stack.a (stack.t) = d;
                    end;
          endx;
     end push;

     pop: procedure (s) integer entry;
          declare s object;
          declare d integer;
          do exclusion;
               if stack.t < n then
                    do;
                         d = stack.a (stack.t);
                         stack.t = stack.t + 1;
                    end;
          endx;
          return d;
     end pop;

end stack;
```

El programa principal USESTACK declara dos objetos A y B a los cuales se les da un identificador con OBJECTID. El método CREATESTACK crea dos instancias de STACK y las asigna a los identificadores A y B. El programa guarda en la pila A los valores impares 1, 3 y 5, y en la pila B los valores pares 2, 4 y 6 usando el método PUSH. Después, escribe el contenido de ambas pilas usando el método POP.

```
$large
usestack: do;
$include (xinu.ext)

     declare (a, b) object;

     createstack: procedure (s) external;
          declare s object;
     end createstack;

     push: procedure (d, s) external;
          declare d integer, s object;
     end push;
```

```
     pop: procedure (s) integer external;
          declare s object;
     end pop;

     do concurrent;
          a = objectid;
          b = objectid;
          call createstack (a);
          call createstack (b);
          call push (1, a);
          call push (2, b);
          call push (3, a);
          call push (4, b);
          call push (5, a);
          call push (6, b);
          call writehex (pop (a), ' ');
          call writehex (pop (a), ' ');
          call writehex (pop (a), cr);
          call writehex (pop (b), ' ');
          call writehex (pop (b), ' ');
          call writehex (pop (b), cr);
     endc;

end usestack;
```

El programa produce lo siguiente:

```
0005 0003 0001
0006 0004 0002
```

Es posible que varios procesos invoquen los métodos de STACK simultaneamente, lo que podría ocasionar errores de dependencia en el tiempo produciendo inconsistencia en los datos. Para evitar esto, las sentencias DO ASSIGN y DO EXCLUSION proveen los mecanismos para asegurar el acceso exclusivo a la región crítica.

La idea principal para representar un objeto en PL/M-86 proviene al observar la analogía entre las reglas de alcance de las variables y procedimientos definidos en un módulo, y la definición de clase. Si las variables no son declaradas públicas, su ámbito se restringe a ese módulo. Los procedimientos públicos son el único medio de accesar a los datos. Los detalles de la implantación permanecen ocultos para el programador y, además, los puede usar en otros programas como tipos adicionales de datos.

La dificultad de representar una clase con un módulo consiste en poder crear varias instancias, cada una con su propio segmento de datos.

Un identificador de clase se representa como un índice del arreglo CLASTAB. Cada identificador tiene asociado una clave que corresponde al segmento de código del módulo que define a la clase. La creación de un

identificador de clase es transparente para el programador, ya que se realiza al momento de aplicar el método de asignación a un objeto.

Un identificador de objeto es un índice de la tabla OBJCTAB. Cada elemento de OBJCTAB se compone de una variable USED que establece si está disponible; IDCL es el identificador de la clase; AREA es el segmento de datos usado exclusivamente por este objeto; SIZE es el tamaño en bytes del area de datos. MUTX y NEXT son semáforos que garantizan el acceso exclusivo a los datos.

OBJECTID devuelve un identificador de objeto disponible. INITCLASS inicia las tablas de las clases y los objetos.

Se debe notar que es posible declarar más de una clase, cada una definida en un módulo diferente, y crear más de un objeto por clase. Darle a un objeto un identificador no presupone que pertenezca a alguna clase. El método de asignación realiza esta función. Los métodos no deben tener los atributos REENTRANT ni INTERRUPT.

Cuando se invoca el método de asignamiento, se ejecuta el calificador DOASSIGN, cuyas acciones son:

1. Obtiene el identificador del objeto y de la clase, examinando los contenidos de la pila de ejecución.

2. Determina el tamaño en bytes de los datos declarados en la clase.

3. Crea un bloque de memoria con el tamaño adecuado para guardar los datos.

4. Cambia el segmento de datos por el segmento del bloque de memoria solicitado.

5. Crea los semáforos MUTX y NEXT, y los inicia con 1 y 0, respectivamente.

Luego de DOASSIGN, se ejecutan las sentencias de la definición del procedimiento que inician las variables globales del objeto. Cambiar el segmento de datos (DS) por el segmento del bloque gestionado, ocasiona que los valores de los datos se conservan para cada objeto.

Para determinar el tamaño exacto del area de datos, la sentencia DO ASSIGN obtiene el valor de la parte desplazamiento de una variable que declara. El compilador asigna las variales del módulo al segmento de datos aún cuando se trate de variables locales. De esta forma el tamaño del area de datos es el desplazamiento de la variable. Por esta razón, es necesario que los datos se declaren primero y a continuación el método de asignación.

DOEXCLUSION se encarga de recuperar el area de datos del objeto a partir de su identificador. Emite una operación WAIT sobre el semáforo MUTX

del objeto (iniciado a 1).

ENDEXCLUSION señala al semáforo MUTX invocando a SIGNAL, para que otro método o proceso pueda usar los datos.

El concepto de clase es muy importante en la programación concurrente. De ella se derivan varias construcciones estructuradas para resolver problemas de sincronización y acceso exclusivo como la región crítica, las regiones críticas condicionales y los monitores.

Un monitor es una clase que permite compartir en forma segura y efectiva los datos requeridos por varios procesos. La principal diferencia semántica con la clase es que sólo un proceso a la vez puede permanecer activo dentro del monitor. Esta propiedad la garantiza el propio monitor usando mecanismos adicionales llamadas variables condición.

Las únicas operaciones que se pueden aplicar sobre las variables condición son CEMPTY, CSIGNAL y CWAIT.

Si C es una variable condición, la operación CWAIT(C) significa que el proceso que la invoca queda suspendido hasta que otro proceso emita la operación CSIGNAL(C).

CSIGNAL reactiva exactamente un proceso suspendido. Si no hay procesos suspendidos, entonces la operación CSIGNAL no tiene efecto, en contraste con la operación SIGNAL que siempre afecta el estado del semáforo.

CEMPTY es una función que reporta el estado de una variable condición. Devuelve el valor TRUE si hay procesos bloqueados esperando sobre la condición.

A continuación mostraremos como se implantaron los monitores usando semáforos.

Las variables condición se representan por un arreglo CONDTAB. Una condición es un elemento de CONDTAB compuesto de una variable USED que indica su disponibilidad; OBID es el identificador del objeto al que pertenece la condición; COND es el semáforo donde los procesos serán bloqueados por CWAIT.

CONDITIONID crea una condición y devuelve su identificador. Determina a qué objeto pertenece la condición y crea el semáforo COND iniciándolo a cero. Se puede declarar más de una variable condición por objeto. Antes de aplicar las operaciones CWAIT, CSIGNAL y CEMPTY sobre una condición se le debe asignar su identificador, usando CONDITIONID en la iniciación de las variables del objeto, es decir, en el método de asignación.

El proceso que emite CWAIT se bloquea e ingresa a una cola de procesos en espera sobre el semáforo COND de la condición. Sin embargo, antes de

bloquearse debe permitir que otro proceso entre al monitor emitiendo una operación SIGNAL sobre el semáforo MUTX del objeto. De otra forma, ningún otro proceso podría entrar al monitor y los bloqueados permanecerían en ese estado indefinidamente.

CSIGNAL activa el primer proceso bloqueado en la condición, si hay alguno. Si no hay procesos bloqueados en la condición, no tiene ningún efecto.

Cuando un proceso P emite la operación CSIGNAL sobre una variable condición, donde había un proceso bloqueado Q, se ha decidido bloquear a P en el semáforo NEXT de la condición. Esto asegura que solo un proceso activo permanece dentro del monitor.

Si el proceso P abandona el monitor o se bloquea sobre una condición, entonces, otro proceso puede activarse (si estaba esperando) o puede entrar. Si hay procesos en espera sobre NEXT, el primero de ellos se activa. Si no hay procesos, se permite la entrada en el monitor aplicando la operación SIGNAL sobre el semáforo MUTX.

El problema clásico de concurrencia "lectores y escritores", demuestra como usar los monitores. Aparece en la sección de aplicaciones.

INTERRUPCIONES

Una interrupción externa es una señal producida por un dispositivo, solicitando servicio del procesador. En atención a dicha solicitud, el procesador ejecuta una rutina de servicio. Esta rutina guarda el estado de la computación en el momento en que ocurrió la interrupción y lo restaura al final del servicio.

Debido a que las rutinas de interrupción accesan y modifican las estructuras de datos del sistema, se debe prevenir que otros procesos interfieran. Una forma de asegurar esto es deshabilitar el sistema de interrupciones del sistema, con lo cual, si el proceso no renuncia al control, sólo él tendrá acceso. Las interrupciones deberán permanecer deshabilitadas hasta que se hayan completado todas los cambios a las esctructuras de datos del sistema.

Sin embargo, el sistema de interrupciones no debe permanecer deshabilitado por mucho tiempo, porque causaría un mal funcionamiento. Por ejemplo, los caracteres trasmitidos por una terminal asíncrona se pierden mientras permanezcan deshabilitadas las interrupciones.

Una consideración muy importante de diseño es que las rutinas de servicio deban funcionar, aun cuando el único proceso en el sistema sea el proceso nulo. Esto significa que la rutina de atención debe dejar al proceso interrumpido en ejecución o como elegible para el uso del CPU. De esta forma, las rutinas pueden usar operaciones como SEND o SIGNAL pero no WAIT.

Finalmente, el cambio de contexto se puede realizar sin problemas si las estructuras de datos del sistema, se actualizan completamente antes del cambio de contexto y que nadie, más que las rutinas de interrupción sea la que rehabilite el sistema de interrupciones.

Para ilustrar cómo se pueden usar estas consideraciones en el procesamiento en tiempo real, se presenta un administrador de reloj. Se

definen nuevos servicios del núcleo de concurrencia como la capacidad de "dormir" a un proceso.

ADMINISTRADOR DE RELOJ

Un reloj en tiempo real significa que un mecanismo de hardware interrumpe al CPU a intervalos regulares durante un segundo. Esta interrupción esta asignada al vector 8 del procesador 8088 y se produce a intervalos de 18.2 "ticks" por segundo. Una rutina del BIOS de la PC atiende normalmente estas señales. Esta rutina produce una interrupción por "software" (01CH) para permitir que el programador escriba sus rutinas de control.

El administrador de reloj CLKINT es la rutina de control de XINU que contiene algunas modificaciones del diseño original para el procesamiento diferido, como se verá mas adelante.

Por las reglas de manejo de interrupciones, si al procesador le toma mucho tiempo atender la interrupción, o si opera con el sistema de interrupciones deshabilitado, entonces el sistema podría tener fallas graves. El sistema usa el tiempo del reloj para limitar la cantidad de tiempo que un proceso puede actuar. Proporciona además, servicios para hacer esperas por un tiempo especificado. Para esto, el sistema mantiene una lista de eventos ordenados por el tiempo en que deben ocurrir. Cada vez que una interrupción del reloj se produce, el sistema examina la lista de eventos y activa a alguno si el tiempo de espera se ha terminado.

El núcleo hace una reasignación ("preemption") para asegurar que los procesos sólo cuenten con el uso del CPU por una cantidad limitada de tiempo. La constante QUANTUM, es la medida del tiempo que el sistema asigna a un proceso.

Cuando un proceso duerme por un tiempo determinado, el sistema lo inserta en una lista ordenada (DELTAQ) por orden creciente de tiempos, no por el orden de su aparición o prioridad. Los procesos en la lista CLOCKQ están ordenados por el tiempo relativo en el cual serán despertados; cada tiempo dice el número de pulsos de reloj que el proceso debe esperar dormido sobre el elemento que le precede en la lista. Con cada pulso del reloj, la clave del primer elemento se decrementa. Cuando llega a cero, el proceso se remueve de la lista y se despierta, es decir, se hace elegible para el uso del CPU.

Dormir a un proceso difiere de suspenderlo en varias formas. En ambos

estados no se compite por el uso del procesador. Un proceso sólo puede ponerse a dormir él mismo, en tanto que un proceso puede suspender a cualquier otro, inclusive, a sí mismo. Ningún proceso puede despertar a otro dormido, solo el administrador del reloj puede hacerlo. Cualquier proceso en ejecución puede reactivar a un proceso suspendido. Dormir es una actividad determinista ya que se fija un tiempo exacto. En cambio, no es posible fijar cuándo se reactivará si estaba suspendido.

El procedimiento SLEEP10 pone a un proceso en estado dormido y se asigna a la lista CLOCKQ por el tiempo especificado en décimas de segundo, apróximadamente. En forma similar, SLEEP causa que el proceso que lo invoca duerma por el tiempo especificado en segundos.

El procesamiento diferido es otra capacidad adicional que ofrece el sistema. El modo diferido permite acumular pulsos de reloj sin iniciar los eventos que normalmente se hubieran despertado. Aquí hay una ligera modificación al diseño original de XINU. Los pulsos de reloj no se acumulan sobre una variable sino que decrementan efectivamente la clave del primer elemento. Cuando esta se vuelve cero, se prosigue a decrementar la clave del siguiente, si queda alguno, repitiéndose esta secuencia hasta que se deshabilite el procesamiento diferido o se agote la lista. Entonces, cuando se reanude la operación normal del reloj, todos los procesos que debieron ser despertados (aquellos con clave igual a cero) se remueven de la lista y se reactivan.

Esta modificación tiene la ventaja de que la duración del procesamiento diferido no se restringe a determinado número de pulsos que puede mantener una variable. Pone en cero la cuenta de todos los procesos que debieron ser despertados pero no los activa. Esto evita las fallas que pueden ocurrir por hacer dormir procesos por mucho tiempo.

El procedimiento STRTDEFCLK, habilita el procesamiento en modo diferido y STOPCLOCK, lo regresa al modo de operación normal o en tiempo real.

Se asume que cuando la lista de procesos dormidos está vacía, el proceso nulo UDI, que corre con la prioridad mas baja, abandona el ciclo repetitivo y restaura el vector de interrupción previo del reloj 01CH. Este es el criterio que determina el fin del ambiente de concurrencia.

INICIACION DEL SISTEMA

La iniciación del núcleo de concurrencia consiste en iniciar las estructuras de datos del sistema y transformar, por si mismo, un solo programa en ejecución en un ambiente capaz de ejecutar varios programas concurrentemente.

Puesto que el PL/M-86 CONCURRENTE está diseñado para correr programas de usuario que requieren ser modificados y probados constantemente, debe ser capaz, igualmente, de detener la ejecución concurrente de varios procesos para convertirse, de nuevo, en un solo programa secuencial.

El procedimiento SYSINIT inicia el sistema de entrada/salida y las estructuras de datos del núcleo, llamando los procedimientos de iniciación que exporta cada uno de los módulos que componen al núcleo. Estos módulos son el de administración de memoria (MEM.P86), manejo de colas con prioridad (Q.P86), administración de procesos (PROC.P86), sincronización con semáforos y mensajes cortos (SEM.P86), comunicación con puertos (PORTS.P86), administración del reloj en tiempo real (CLOCK.P86) y tipos abstractos de datos (DATA.P86).

SYSEND crea un proceso especial llamado UDI, que corresponde al programa en ejecución, antes y después del ambiente de concurrencia, es decir, el subsistema UDI y el sistema operativo MSDOS. Este proceso especial corre con la prioridad más baja posible, de manera que cualquier otro proceso permanecerá funcionando hasta que sea destruido o permanentemente bloqueado. El código, el area de datos, la pila de ejecución y los registros del procesador de UDI se guardan en el primer cambio de contexto y se recuperan en el último.

El proceso UDI simplifica la administración de procesos y se identifica con el proceso nulo del diseño original de XINU. Es un proceso que se autodestruye cuando no hay más procesos elegibles para recibir el CPU. Se representa por un ciclo que se repite mientras haya al menos otro proceso activo, esperando mensaje o esperando en la lista de dormidos. La ejecución concurrente termina cuando no hay mas procesos elegibles ni

procesos dormidos.

Se debe recordar que las llamadas a SYSINIT y SYSEND, están ocultas en la proposición DO CONCURRENT...ENDC. SYSINIT se invoca al principio de la proposición, y SYSEND al final.

Por ejemplo, si PRODUCER y CONSUMER son dos procesos:

```
do concurrent;
     call producer;
     call consumer;
endc;
```

se producirá la siguiente secuencia de invocaciones:

```
call sysinit;
     call producer;
     call consumer;
call sysend;
```

Cuando el control alcanza el final de la sentecia DO CONCURRENT es cuando en realidad comienza la ejecución concurrente, al realizarse el primer cambio de contexto, RESCHEDULE (que llama a CTXSW).

SYSEND establece el valor de las variables globales CLKRUNS y CURRPID.

Aun cuando el administrador del reloj ya fué instalado por SYSINIT, no empieza su funcionamiento hasta que CLKRUNS es puesto a TRUE.

Al hacer que el proceso en ejecución sea el proceso nulo (CURRPID = NULLPROC), se garantiza que el ambiente del sistema operativo DOS quede sin cambio y sea recuperado en la misma forma.

Cuando termina la ejecución concurrente, SYSEND deshabilita el administrador del reloj poniendo CLKRUNS a FALSE. Después, reasigna los vectores de interrupción previos que había tomado para el reloj y para CTXSW.

Durante la ejecución del programa, se pueden producir errores en el funcionamiento del núcleo. El procedimiento SYSFAIL genera un mensaje de error y trata de abortar la ejecución concurrente, forzando a que el proceso UDI entre en ejecución. El éxito de esta medida correctiva depende del tipo de error producido. En el apéndice III aparece la lista de errores del núcleo producidos a tiempo de ejecución.

CAPITULO IV

APLICACIONES

PROBLEMAS CLASICOS DE CONCURRENCIA

En este capítulo, daremos varios ejemplos que muestran como escribir programas que involucran concurrencia usando el lenguaje PL/M-86 CONCURRENTE. El lector interesado en una discusión mas detallada puede consultar la bibliografía que se incluye.

Estos programas presentan una solución a diferentes problemas de sincronización que son importantes porque son representativos de una gran clase de problemas de control de procesos. Los problemas clásicos se usan para probar un nuevo esquema de sincronización como el propuesto para los mensajes compartidos. En su escritura usaremos las clases, los semáforos, los monitores o la comunicación con puertos como elementos de sincronización.

LA RELACION PRODUCTOR-CONSUMIDOR USANDO CLASES

El problema de la relación productor-consumidor [Peterson y Silverschatz 1981] es muy común en las aplicaciones de la programación concurrente, sobre todo en los sistema operativos. Este problema consiste en que existe un proceso productor de datos y un proceso consumidor de los mismos. Por ejemplo, un programa puede producir caracteres que deben ser consumidos por la impresora.

Para que ambos procesos puedan operar correctamente se necesita sincronizar sus acciones, ya que el consumidor no puede consumir datos que el productor no ha producido. Por otra parte, el productor puede producir datos que tarden en ser consumidos. El problema se resuelve considerando un área de espera donde los datos serán depositados y obtenidos, representados

por una clase QUEUE que realice estas dos operaciones básicas. Suponemos que el tamaño del área está limitada.

El programa se compone de dos módulos, QUEUE que declara a la clase y PRODUCERCONSUMER que es el programa principal. Ambos módulos son compilados por separado con el modelo de segmentación largo y unidos en la fase de encadenamiento.

```
$large
queue: do;
$include (class)

      declare n literally '10';
      declare queue structure ((head, tail, coun) byte, area (n) byte);

      cqueue: procedure (q) entry;
            declare q object;
            do assign;
                  queue.head, queue.tail, queue.coun = 0;
            enda;
      end cqueue;

      qadd: procedure (d, q) entry;
            declare d byte, q object;
            do exclusion;
                  if queue.coun < n then
                        do;
                              queue.coun = queue.coun + 1;
                              queue.tail = (queue.tail + 1) mod n;
                              queue.area (queue.tail) = d;
                        end;
            endx;
      end qadd;

      qdel: procedure (q) byte entry;
            declare q object;
            declare d byte;
            do exclusion;
                  if queue.coun > 0 then
                        do;
                              queue.coun = queue.coun - 1;
                              queue.head = (queue.head + 1) mod n;
                              d = queue.area (queue.tail);
                        end;
            endx;
            return d;
      end qdel;
end queue;
```

La clase QUEUE declara los datos y define los métodos de acceso usados

en el programa principal. Las definiciones de las clases y los monitores están contenidas en el archivo CLASS. QUEUE implementa una cola circular con representación contigua, con apuntadores al primer y último elementos y un contador de elementos. El atributo ENTRY indica que el procedimiento es un método para accesar los datos que, de otra forma, sería un procedimiento local al módulo.

El procedimiento CQUEUE contiene una sentencia DO ASSIGN la cual crea una instancia del objeto en el momento de su invocación y lo asigna al identificador de objeto Q de la lista de parámetros. Al mismo tiempo, inicia las variables globales del módulo. CQUEUE debe ser el primer método que se invoque en el programa principal.

Los procedimientos QADD y QDEL implementan las operaciones básicas sobre la cola, de adición y eliminación de elementos. Puesto que puede haber más de una instancia de la clase QUEUE, se debe informar sobre qué identificador de objeto se va a aplicar la operación. El identificador debe aparecer invariablemente como último elemento de la lista de parámetros. La sentencia DO EXCLUSION realiza este encadenamiento dinámico entre el identificador de objeto y el área de datos asignada. Proporciona, además, el acceso exclusivo a los datos de la clase, asegurando que nunca ocurrirá un acceso simultáneo de QADD y QDEL a un objeto de QUEUE.

Las sentencias DO ASSIGN y DO EXCLUSION deben aparecer sólo una vez y encontrarse en el nivel más externo de la definición del procedimiento.

```
$large
producerconsumer: do;
$include (xinu.ext)

     declare q object, (p, c) semaphore;

     cqueue: procedure (q) external;
          declare q object;
     end cqueue;

     qadd: procedure (d, q) external;
          declare d byte, q object;
     end qadd;

     qdel: procedure (q) byte external;
          declare q object;
     end qdel;

     producer: procedure (q, p, c) reentrant;
          declare q object, (p, c) semaphore;
          declare d byte;
          do process;
               do d = 0 to 127;
                    call wait (p);
```

```
                    call qadd (d, q);
                    call signal (c);
               end;
          endp;
     end producer;

     consumer: procedure (q, p, c) reentrant;
          declare q object, (p, c) semaphore;
          declare d byte;
          do process;
               do loop;
                    call wait (c);
                    d =  qdel (q);
                    call signal (p);
                    call writechr (d);
               end;
          endp;
     end consumer;

     do concurrent;
          q = objectid;
          p = screate (n);
          c = screate (0);
          call cqueue (q);
          call producer (q, p, c);
          call consumer (q, p, c);
     endc;
end producerconsumer;
```

El programa PRODUCERCONSUMER declara un objeto Q al que se le da un identificador mediante OBJECTID y se asigna a la clase QUEUE invocando al método CQUEUE. Los métodos de la clase se declaran como externos. Los semáforos C y P son creados e iniciados por SCREATE y se usan para sincronizar las acciones de los procesos CONSUMER y PRODUCER.

La sentencia DO PROCESS califica a los procedimientos CONSUMER y PRODUCER como procesos que son creados en el momento de su invocación. Puesto que cada proceso en PL/M-86 CONCURRENTE posee su propia area de datos una forma de pasar las variables declaradas en el programa es como parámetros del procedimiento. DO PROCESS debe aparecer en el nivel más externo de la definición del procedimiento y sólo una vez.

La sentencia DO CONCURRENT crea el ambiente de concurrencia y activa los procesos creados. Esta sentencia sigue las mismas reglas de uso de DO PROCESS excepto que se aplica al cuerpo del programa principal.

El proceso PRODUCER genera una secuencia creciente de enteros que está limitada por el semáforo P. Cada vez que un dato se produce, la cuenta del semáforo C aumenta indicando a CONSUMER que ya puede reanudar su labor.

LA RELACION PRODUCTOR-CONSUMIDOR USANDO MENSAJES

Una solución alternativa al problema del productor-consumidor, puede plantearse en términos de las operaciones de comunicación entre procesos. El programa CONSUMERPRODUCER utiliza los mensajes estructurados para trasmitir los datos generados por el proceso productor al proceso consumidor.

```
$large
consumerproducer: do;
$include (xinu.ext)

     declare n literally '10';
     declare p port, d byte;

     producer: procedure (p) reentrant;
          declare p port;
          declare d byte;
          do process;
               do d = 0 to 127;
                    call psend (p, @d);
               end;
          endp;
     end producer;

     consumer: procedure (p) reentrant;
          declare p port;
          declare d byte;
          do process;
               call preceive (p, @d);
               do loop;
                    call writechr (d);
                    call preceive (p, @d);
               end;
          endp;
     end consumer;

     do concurrent;
          p = pcreate (n, sizeof (d));
          call producer (p);
          call consumer (p);
     endc;
end consumerproducer;
```

La variable P contiene el identificador del puerto usado para el intercambio de mensajes entre los procesos. La operación PCREATE crea un

puerto con capacidad de N mensajes, de SIZEOF(D) bytes de longitud por cada mensaje. La variable D sólo sirve para informar a PCREATE el tamaño de la estructura de los mensajes. Aun cuando en este ejemplo la estructura de los mensajes es muy sencilla (un byte), las operaciones sobre puertos se aplican igualmente a estructuras arbitrariamente complejas.

PRODUCER genera una secuencia de enteros enviada al puerto P por PSEND. Aun cuando se requiere la dirección del dato, éste es en realidad enviado por copia y no por referencia. Una vez agotada la capacidad del puerto, PRODUCER es forzado a detenerse hasta que, eventualmente, llega ha haber más espacio.

El proceso CONSUMER demanda datos invocando a PRECEIVE. Cuando hay mensajes disponibles en el puerto, se hace una copia y se envia a CONSUMER. De otra forma, el proceso espera hasta que aparezca algún dato sobre el puerto.

Una breve comparación entre las soluciones propuestas por semáforos y por mensajes demuestra la superioridad de ésta última en problemas de comunicación de procesos. Las operaciones sobre puertos evitan las equivocaciones que pueden resultar del manejo directo con semáforos, adicionalemente al intercambio de información entre procesos.

LOS FILOSOFOS

Este problema fué enunciado y resuelto por Dijkstra [1965]. El problema puede enunciarse de la siguiente forma: Hay cinco filósofos que pasan su vida pensando y comiendo. Comparten una mesa rodeada de cinco sillas, cada una de las cuales pertenece a un filósofo. En el centro de la mesa hay una platón de arroz y cinco palillos alrededor. Cuando un filósofo tiene hambre, trata de tomar los dos palilos que le queden mas cerca. Sólo si tiene ambos, puede empezar a comer. Una vez que ha terminado, pone los palillos en la mesa de nuevo y se pone a pensar. Cuando el filósofo está pensando, no compite por palillos.

La presente solución del problema se debe a Ben-Ari [1982]. El escenario consiste en un comedor en donde caben hasta cuatro filósofos que deseen comer. Cualquier intento de distribuir los cinco palillos entre los cuatro filósofos resulta en que siempre hay un filósofo con dos palillos. Cuando termina de comer un filósofo, sale del comedor a pensar.

```
$large
diningphilosophers: do;
$include (xinu.ext)
```

```
      declare fork (5) semaphore, room semaphore, i integer;

      think: procedure (i);
           declare i byte;
           call writestr (@ ('Filosofo ',0), i + '0');
           call writestr (@ (' pensando...',0), cr);
           call sleep10 (signed (rand (100)));
      end think;

      eat: procedure (i);
           declare i byte;
           call writestr (@ ('Filosofo ',0), i + '0');
           call writestr (@ (' comiendo...',0), cr);
           call sleep10 (signed (rand (100)));
      end eat;

      philosopher: procedure (i, room, forkp) reentrant;
           declare i integer, room semaphore, forkp pointer;
           declare fork based forkp (*) semaphore;
           do process;
                do loop;
                     call think (i);
                     call wait (room);
                     call wait (fork (i));
                     call wait (fork ((i + 1) mod 5));
                     call eat (i);
                     call signal (fork (i));
                     call signal (fork ((i + 1) mod 5));
                     call signal (room);
                end;
           endp;
      end philosopher;

      do concurrent;
           room = screate (4);
           do i = 0 to 4;
                fork (i) = screate (1);
           end;
           do i = 0 to 4;
                call philosopher (i, room, @fork);
           end;
      endc;
end diningphilosophers;
```

Los palillos y el comedor se representan con semáforos. PHILOSOPHER implanta los cinco procesos que comparten el mismo código, pero cada uno con su propia area de datos. Los cinco procesos se crean con su invocación y teniendo como identificador el argumento I de su lista de parámetros.

LECTORES Y ESCRITORES

Este problema fue enunciado y resuelto por Courtois [1971]. Consiste en que varios procesos concurrentes comparten objetos (como los registros de una base de datos) y que es preciso evitar los resultados adversos que esto acarrea. Algunos procesos realizan solo la lectura del objeto compartido, mientras otros lo pueden modificar. Los procesos lectores pueden accesar simultaneamente el objeto sin ningún peligro. En cambio, el acceso por parte de mas de un proceso escritor puede ocasionar resultados contradictorios. Este problema tiene variaciones para su solución, entre las cuales escogimos la propuesta por Ben-Ari [1982] fijada por las siguientes reglas:

1. Si hay procesos escritores esperando, entonces cualquier nuevo lector debe esperar a que concluyan todos los escritores.

2. Si hay lectores esperando la terminación de la escritura, ellos tienen prioridad sobre otro escritor que arribe.

El escenario consiste de una sala de consulta a la cual asisten los lectores y escritores sujetos a las reglas establecidas. El monitor READWRITE asegura el uso consistente de los objetos compartidos por ambos tipos de procesos.

```
$large
readwrite: do;
$include (class)

     declare readers integer, writing byte;
     declare (oktoread, oktowrite) condition;

     initreadwrite: procedure (lib) entry;
          declare lib object;
          do assign;
               readers = 0;
               writing = false;
               oktoread = conditionid;
               oktowrite = conditionid;
          enda;
     end initreadwrite;

     beginread: procedure (lib) entry;
          declare lib object;
          do exclusion;
               if writing or not cempty (oktowrite)
                    then call cwait (oktoread);
```

```
                readers = readers + 1;
                call csignal (oktoread);
           endx;
      end beginread;


      endread: procedure (lib) entry;
           declare lib object;
           do exclusion;
                readers = readers + 1;
                if readers = 0 then call csignal (oktowrite);
           endx;
      end endread;

      beginwrite: procedure (lib) entry;
           declare lib object;
           do exclusion;
                if readers <> 0 or writing then call cwait (oktowrite);
                writing = true;
           endx;
      end beginwrite;

      endwrite: procedure (lib) entry;
           declare lib object;
           do exclusion;
                writing = false;
                if not cempty (oktoread) then call csignal (oktoread);
                else call csignal (oktowrite);
           endx;
      end endwrite;
end readwrite;
```

El procedimiento INITREADWRITE crea una nueva instancia del monitor, inicia las variables globales y asigna dicha instancia a un identificador de objeto a tiempo de ejecución. La variable READERS lleva la cuenta de los lectores que se encuentran en la sala de consulta. La condición OKTOREAD significa una cola de procesos lectores en espera, de donde serán obtenidos por CSIGNAL. La variable WRITING indica si hay un escritor actualizando el objeto. El proceso escritor que está listo para entrar espera en la condición OKTOWRITE.

El programa READERSANDWRITERS da la solución completa al problema. El programa y el monitor son unidades que deben compilarse separadamente y unidas en el encadenamiento.

Tanto la clase QUEUE como el monitor READWRITE utilizan las mismas sentencias DO ASSIGN y DO EXCLUSION. La diferencia entre una clase y un monitor en PL/M-86 CONCURRENTE es que este último utiliza variables de tipo CONDITION junto con sus operaciones CWAIT, CSIGNAL y CEMPTY.

```
$large
readersandwriters: do;
$include (xinu.ext)

     readwrite: procedure (lib) external;
          declare lib object;
     end readwrite;

     beginread: procedure (lib) external;
          declare lib object;
     end beginread;

     endread: procedure (lib) external;
          declare lib object;
     end endread;

     beginwrite: procedure (lib) external;
          declare lib object;
     end beginwrite;

     endwrite: procedure (lib) external;
          declare lib object;
     end endwrite;

     waitingread: procedure;
          call writestr (@ ('Lector espera entrar',0), cr);
          call sleep10 (100);
     end waitingread;

     waitingwrite: procedure;
          call writestr (@ ('Escritor espera entrar',0), cr);
          call sleep10 (100);
     end waitingwrite;

     consult: procedure;
          call writestr (@ ('Lector entra',0), cr);
          call sleep10 (100);
          call writestr (@ ('Lector sale',0), cr);
     end consult;

     update: procedure;
          call writestr (@ ('Escritor entra',0), cr);
          call sleep10 (100);
          call writestr (@ ('Escritor sale',0), cr);
     end update;

     reader: procedure (library);
          declare library object;
          do process;
               do loop;
```

```
                    call waitingread;
                    call beginread (library);
                    call consult;
                    call endread (library);
               end;
          endp;
     end reader;

     writer: procedure (library);
          declare library object;
          do process;
               do loop;
                    call waitingwrite;
                    call beginwrite (library);
                    call update;
                    call endwrite (library);
               end;
          endp;
     end writer;

     do concurrent;
          library = objectid;
          call initreadwrite (library);
          call reader (library);
          call reader (library);
          call writer (library);
          call writer (library);
     endc;
end readersandwriters;
```

LOS FUMADORES

Este problema se debe a Patil [1971]. El escenario consiste en tres fumadores y un agente. Cada fumador hace constantemente un cigarro y se lo fuma. Sin embargo, para hacer un cigarro se requieren tres ingredientes: tabaco, papel y cerillos. Un fumador tiene tabaco, otro papel y otro cerillos. El agente tiene una cantidad ilimitada de los tres y coloca dos ingredientes distintos en una mesa. El fumador que complete los tres ingredientes puede hacer y fumar el cigarro, indicando al agente cuando termine. Entonces, el agente coloca otros dos ingredientes y se repite el ciclo.

En esta solución utilizo la técnica desarrollada en este trabajo de los "mensajes compartidos". El lector puede comparar esta solución con la mostrada por Ben-Ari [1982].

```
declare cigarret literally 'structure ((a, b) byte)';
declare table port, ingredient cigarret;

product: procedure (p);
     declare p byte;
     do case (p);
          call writestr(@('tabaco,',0),' ');
          call writestr(@('papel,',0),' ');
          call writestr(@('cerillos,',0),' ');
     end;
end product;

agent: procedure (table) reentrant;
     declare table port;
     declare ingredient cigarret;
     do process;
          do loop;
               ingredient.a = rand (3);
               do while (ingredient.a <> (ingredient.b := rand (3)));
               end;
               call pwrite (table, @ingredient);
               call writestr (@ ('Agente produce',0), ' ');
               call product (ingredient.a);
               call product (ingredient.b);
          end;
     endp;
end agent;

smoker: procedure (table, i) reentrant;
     declare table port, i byte;
     declare ingredient cigarret;
     do process;
          do loop;
               call pread (table, @ingredient);
               if (ingredient.a <> i) and (ingredient.b <> i) then
                    do;
                         call writestr (@('Fumador con',0),' ');
                         call product (i);
                         call product (ingredient.a);
                         call product (ingredient.b);
                         call writestr (@('disfruta su cigarro.',0),cr);
                    end;
          end;
     endp;
end smoker;

do concurrent;
     table = pcreate (1, sizeof (ingredient));
     call agent (table);
     call smoker (table, 0);
```

```
        call smoker (table, 1);
        call smoker (table, 2);
        call pcontroller (table);
    endc;
end smokers;
```

Puesto que cada uno de los tres fumadores posee un ingrediente distinto, la estructura del cigarro está formada por los dos componentes que le faltan. Todos los mensajes poseen esta estructura. La mesa donde el agente deposita los ingredientes es el puerto de mensajes compartidos por cada fumador. TABLE contiene el identificador del puerto creado por PCREATE con un mensaje de capacidad y con un tamaño de mensaje de 2 bytes.

El proceso AGENT produce aleatoriamente dos ingredientes distintos depositados en TABLE por PWRITE. Puesto que el puerto sólo es de un mensaje de capacidad, AGENT es forzado a esperar.

Los tres procesos SMOKER examinan el ingrediente depositado en la mesa usando PREAD. PREAD bloquea siempre al proceso.

En este punto, el agente y los tres fumadores se hayan bloqueados. Entonces, se activa el proceso PCONTROLLER con menor prioridad para desbloquear los procesos que se encuentren sobre el puerto.

Cuando termina de activar los procesos, forza un cambio de contexto y les cede el control.

Luego, los fumadores examinan el ingrediente. Sólo uno de ellos puede completar su cigarro y fumarlo. Cuando lo termina intentará, como los otros fumadores, formar un cigarro.

El agente produce otro ingrediente y se repite el ciclo.

Este problema es representativo de muchas aplicaciones.

APLICACIONES

PASCAL CONCURRENTE

A continuación se describirá como llevar la técnica de los calificadores a otros lenguajes como Pascal, con la creación de rutinas de servicio de interrupción. Siguiendo las ideas de este trabajo, se definirá la sentencia adicional IOBEGIN/IOUNTIL, que califica a un procedimiento como una rutina de servicio. La sentencia determina del contexto la información necesaria, haciendo transparente para el programador no sólo la instalación de la rutina sino también su activación.

Aun cuando no se revisan los temas de cambio de contexto y creación de procesos, las ideas que se presentan pueden sugerir como realizar una versión del Pascal Concurrente. El lenguaje empleado es el Turbo Pascal 3.0 por su flexibilidad y facilidad de uso.

El lenguaje Turbo Pascal no provee los medios para crear e instalar rutinas de servicio de interrupción como PL/M-86. Sin embargo, posee la capacidad de generación de código en linea por medio de la sentencia INLINE. Otra característica útil es su facilidad para hacer gestiones de memoria de bajo nivel. El procedimiento GETMEM obtiene un bloque de memoria del tamaño especificado en bytes. FREEMEM libera el bloque cuya dirección es la obtenida por GETMEM.

Cualquier localidad de memoria puede accesarse usando los arreglos predefinidos MEM de tipo byte y MEMW de tipo entero. El índice de un elemento de este arreglo coincide con la dirección de una localidad de memoria formada por una parte segmento base y un desplazamiento. La función de tipo entero SEG devuelve el segmento al que está referida una variable o procedimiento y OFS devuelve el desplazamiento relativo a este segmento. Las funciones CSEG, DSEG y SSEG obtienen, respectivamente, el segmento de código, el de datos y el de la pila de ejecución. La función PTR crea una dirección completa de memoria a partir de un segmento y un desplazamiento. Se pueden realizar operaciones sobre los puertos del procesador mediante los arreglos predefinidos PORT de tipo byte y PORTW de tipo entero, cada uno de 64 K elementos.

El lenguaje permite interactuar con el sistema operativo definiendo

una estructura especial, cuyos componentes representan a los registros del procesador, y usando el procedimiento MSDOS.

```
type registers8086 =
        record
           ax, bx, cx, dx, bp, si, di, ds, es, flags : integer
        end;
```

Cualquier instancia de REGISTERS8086 puede usarse para establecer el valor de los registros del procesador antes de la llamada a MSDOS, así como determinar los valores que regresa. Por ejemplo, la función $35 (hex) devuelve la dirección de la rutina de servicio de una interrupción. Antes de la llamada a MSDOS, el registro AH contiene el valor de la función, AL el número de interrupción. El vector de interrupción lo devuelve en el par de regitros ES para el segmento y BX para el desplazamiento. De igual forma, la función $25 instala el vector de interrupción formado por los registros DS y DX.

El diseño de los calificadores en Pascal estará dirigido por las características del lenguaje PL/M-86 para tratar con interrupciones. En PL/M-86, el atributo INTERRUPT en un procedimiento sin tipo ni argumentos, genera el código adecuado en el prólogo y en el epilógo para guardar y restaurar el ambiente previo a la interrupción. El calificador IOBEGIN proverá la instalación de la rutina y su prólogo. IOUNTIL dará el epílogo de la rutina de servicio hasta que se cumpla alguna condición y, entonces, la desinstalará.

Puesto que la rutina de servicio debe guardar el contenido de todos los registros del procesador (excepto SS y SP), la estructura de la pila de ejecución es diferente a la de un procedimiento normal. Por esta razón, no se puede insertar el código que guarde los registros como primera sentencia de la rutina, ya que se puede destruir el contenido de la pila al comenzar la ejecución. Para que esto no ocurra, se debe substituir la ejecución del prólogo usual de un procedimiento por el prólogo adecuado para interrupciones.

IOBEGIN define una plantilla que contiene la estructura del prólogo. Al instalar la rutina de servicio, se hace una copia de esta plantilla en un bloque de memoria. La dirección de este bloque es la dirección de la rutina de servicio. La plantilla debe incluir código para ceder el control al procedimiento del programador. Puesto que no se conoce de antemano esa referencia, se debe resolver a tiempo de compilación o de ejecución. El calificador IOBEGIN la determina a tiempo de ejecución, modificando el bloque. Además de esta dirección, determina otros valores importantes como el segmento de datos, el espacio que ocupan las variables locales y el número de interrupción.

El modelo de segmentación de memoria de Turbo Pascal corresponde al caso compacto de PL/M-86, es decir, hay cuatro segmentos asignados al código, los datos, la pila de ejecución y la gestión de memoria, cada uno

con un tamaño máximo de 64K bytes. Puesto que los segmentos de código, datos y la pila se fijan en un principio y no se modifican, los registros correspondientes CS, DS y SS no se alteran. Sin embargo, cuando una interrupción ocurre el registro DS puede cambiar. Es importante que el prólogo de la rutina de servicio restablezca el valor del registro DS porque las variables globales del programa están referidos a él.

Los parámetros actuales y las variables locales de un procedimiento son asignados a la pila de ejecución y referidos al registro BP. Si el prólogo de la interrupción sólo guarda el contenido de los registros, entonces el procedimiento no puede declarar variables locales porque ocuparían el espacio de los registros en la pila. Para evitar esto, el registro BP debe apuntar al último valor insertado y el registro SP debe ser modificado para crear espacio para las variables locales. Estas acciones aseguran que los valores de los registros y variables locales no serán destruidos. Se debe subrayar que las estructuras de la pila de un procedimiento normal y el de una interrupción son diferentes, por lo cual, no deben hacerse referencias a los parámetros ya que se puede destruir el contenido de los registros en la pila. La rutina de servicio sólo debe incluir un parámetro de tipo entero que es la interrupción a la que se va a asignar la rutina. Este parámetro es informativo y no debe emplearlo el programador.

Finalmente, el prólogo determina la interrupción en curso y la asigna a la variable global INTNUM_. Esta información puede ser útil para conocer qué agente externo produjo la señal. Después de la instalación, la rutina de servicio no permanece residente en memoria.

El calificador IOUNTIL proporciona el epílogo de la rutina de servicio. El epílogo es idéntico en todas las rutinas de servicio y puede insertarse al final del procedimiento. Consiste en señalar al controlador de interrupciones que se ha llegado al final de la interrupción y en restaurar el contenido de los registros. Con el objeto de ocultar estas acciones al programador, se han encapsulado dentro del calificador. Siguiendo las mismas ideas de diseño del PL/M-86 CONCURRENTE, el ciclo de desarrollo de una aplicación requiere de continuas pruebas y modificaciones, siendo conveniente restablecer el ambiente de la programación. Por esta razón, la rutina de servicio se desinstala a sí misma después de haber alcanzado cierta condición. Se debe tener cuidado con las variables locales del procedimiento al evaluar la condición ya que mantienen las mismas reglas de alcance. Si se desea que la rutina quede permanentemente instalada, la condición de desintalación debe ser FALSE.

El siguiente programa muestra cómo se define una rutina de servicio. Esta rutina representa un reloj muy sencillo, asignado al vector de interrupción 1CH ("time of day") y se activa cada vez que el reloj produce un pulso, aproximadamente, 18 veces por segundo.

```
{R+}
{K-}
```

```
program iohandler;

{$I IOHandler}

     var tick, seg, min : integer;

     procedure clock (intnum : integer);
          type digits = string[10];

          function dig2 (num : integer) : digits;
               var s : digits;
          begin
               str (100 + num:3, s);
               delete (s, 1, 1);
               dig2 := s;
          end;

     begin
          iobegin;
               tick := (tick + 1) mod 18;
               if tick = 0 then
                    begin
                         seg := (seg + 1) mod 60;
                         if seg = 0 then
                              min := (min + 1) mod 60;
                         gotoxy (75, 1);
                         write (dig2 (min), ':', dig2 (seg));
               end;
          iountil (min >= 1);
     end;

begin
     tick := 0; seg := 0; min := 0;
     clock ($1C);
     clrscr;
end.
```

La rutina de servicio CLOCK es un procedimiento con el único parámetro de tipo entero INTNUM. Usa las variables globales TICK, SEG y MIN, para llevar la cuenta de los pulsos del reloj, los segundos y minutos, respectivamente. Define varios objetos locales: el tipo DIGITS, la función DIG2 y la variable S (local a DIG2). Las sentencias IOBEGIN / IOUNTIL aparecen en el nivel mas externo del procedimiento y sólo una vez.

Después de iniciar los datos globales, la rutina queda instalada simplemente con su invocación. El argumento $1C informa la interrupción que va a atender. Al igual que el calificador DO PROCESS del PL/M-86 CONCURRENTE, la única sentencia del procedimiento CLOCK que se ejecuta en la invocación es IOBEGIN, que realiza la instalación, cediendo el control

de inmediato a CLRSCR en el programa principal.

Después de su instalación, sólo las señales del reloj activan a CLOCK. La rutina será desinstalada después de transcurrido un minuto, como lo indica la condición de IOUNTIL.

La nueva sentencia estructurada para el manejo de interrupciones del lenguaje Turbo Pascal es flexible, fácil de aprender y de usar. Posee las ventajas ya discutidas de los calificadores.

Discutiremos brevemente las desventajas de esta implantación en particular. La principal es la eficiencia. El lector habrá notado que se realizan operaciones que no son necesarias como en IOUNTIL donde sería mas adecuado insertar el epílogo de la interrupción que llamar un procedimiento que lo ejecute.

Esta consideración es muy importante sobre todo en el procesamiento en tiempo real donde se requiere un rápida respuesta del sistema. Sin embargo, el diseño está orientado para el proceso de desarrollo de prototipos, o bien, en una amplia variedad de situaciones en las cuales la sobrecarga introducida no es significativa, por ejemplo, en el manejo de terminales asíncronas, "spoolers" para impresoras, dispositivos de interface con el hombre como teclado, "joystick", etc.

Un programador que deba realizar una rutina de servicio en un sistema que requiera una rápida respuesta, no pensará en la posiblidad de desarrollarla en un lenguaje de alto nivel, a menos que esté convencido de la eficiencia del código que genera. Pero aún en tales circunstancias, este trabajo sería conveniente para el desarrollo y prueba de un prototipo.

```
{ IOHandler.pas }

  type registers8086 =
         record
            ax, bx, cx, dx, bp, si, di, ds, es, flags : integer;
         end;

   var intnum_  : byte;
       bp_, ip_ : integer;
       inttab_ : array [0..255] of
          record
             cs, ip : integer
          end;

{ Calificador inicial: instala la rutina de servicio de interrupcion }
   procedure iobegin;
      const prologue : array [0..34] of byte =
         ($00, $00,                { [00]  DW    0              }
          $00, $00,                { [02]  DW    0              }
          $51,                     { [04]  PUSH  CX             }
```

```
        $50,                          { [05]   PUSH  AX
        $52,                          { [06]   PUSH  DX
        $53,                          { [07]   PUSH  BX
        $56,                          { [08]   PUSH  SI
        $57,                          { [09]   PUSH  DI
        $06,                          { [0A]   PUSH  ES
        $1E,                          { [0B]   PUSH  DS
        $55,                          { [0C]   PUSH  BP
        $8B, $EC,                     { [0D]   MOV   BP, SP
        $B8, $00, $00,                { [0F]   MOV   AX, <DATA SEGM>
        $8E, $D8,                     { [12]   MOV   DS, AX
        $81, $EC, $00, $00,           { [14]   SUB   SP, <VARS LOCS>
        $B0, $00,                     { [18]   MOV   AL, <INT NUM>
        $A2, $00, $00,                { [1A]   MOV   [INTNUM_], AL
        $FB,                          { [1D]   STI
        $2E, $FF, $2E, $00, $00); { [1E]   JMP   FAR CS:[0000]

    var handler : ^integer;
        hseg : integer; intno : byte;
        registers : registers8086;

  begin
    inline($2E/ $89/ bp_);            {     MOV   [BP_], BP     }
    ip_ := memw[sseg : bp_ + 2];
    intno := mem[sseg : memw[sseg : bp_] + 4];
    getmem(handler, 35);
    move(prologue, handler^, 35);
    hseg := seg(handler^);
    with registers do
      begin
        ax := $3500 + intno;
        msdos(registers);
        inttab_[intno].ip := bx;
        inttab_[intno].cs := es;
        memw[hseg : $00] := ip_;
        memw[hseg : $02] := cseg;
        memw[hseg : $10] := dseg;
        memw[hseg : $16] := memw[sseg : bp] - bp - 8;
        mem [hseg : $19] := intno;
        memw[hseg : $1b] := ofs(intnum_);
        ax := $2500 + intno;
        ds := seg(handler^);
        dx := 4;
        msdos(registers);
      end;
    ip_ := memw[sseg : memw[sseg : bp_] + 2];
    inline($26/ $FF/ ip_);            {      JMP      [IP_]      }
  end;

{ Restaura el vector de interrupcion previo a la instalacion }
```

```
  procedure iorestore;
      var registers : registers8086;
          handler : ^integer;
   begin
      with registers do
         begin
            ax := $3500 + intnum_;
            msdos(registers);
            handler := ptr(es, bx);
            freemem(handler, 30);
            ax := $2500 + intnum_;
            dx := inttab_[intnum_].ip;
            ds := inttab_[intnum_].cs;
            msdos(registers);
         end;
   end;

{ Calificador final: regresa de la rutina de servicio hasta que la condicion
  sea verdadera. Entonces, desinstala la rutina }
   procedure iountil(b : boolean);
   begin
      if b then
         iorestore;
      inline(
         $B0/ $20/                        {   MOV   AL, 20H    }
         $E6/ $20/                        {   OUT   20H, AL    }
         $8B/ $E5/                        {   MOV   SP, BP     }
         $5D/                             {   POP   BP         }
         $8B/ $E5/                        {   MOV   SP, BP     }
         $5D/                             {   POP   BP         }
         $1F/                             {   POP   DS         }
         $07/                             {   POP   ES         }
         $5F/                             {   POP   DI         }
         $5E/                             {   POP   SI         }
         $5B/                             {   POP   BX         }
         $5A/                             {   POP   DX         }
         $58/                             {   POP   AX         }
         $59/                             {   POP   CX         }
         $CF                              {   IRET             }
      );
   end;
```

CONCLUSIONES

PL/M-86 CONCURRENTE es un lenguaje estructurado para la programación de sistemas concurrentes. Es una extensión del lenguaje secuencial PL/M-86. La extensión consiste en agregar un conjunto de estructuras de datos y procedimientos tomados del núcleo del sistema operativo XINU.

Uno de los aspectos más importantes del lenguaje es la modularidad que se obtiene en el diseño de programas. Un sistema escrito en este lenguaje se construye en dos clases de módulos: procesos y clases.

Los procesos son unidades que se ejecutan secuencialmente y no comparten variables. Las clases son tipos abstractos de datos accesados sólo por un proceso a la vez. Un monitor es un tipo especial de clase usado para sincronizar procesos.

Posee una amplia variedad de mecanismos de comunicación. La comunicación entre procesos se consigue por pasaje de parámetros del proceso, comunicación directa (mensajes de tipo entero) y comunicación indirecta (mensajes de tipo definido por el programador). En esta última se definen nuevas operaciones (mensajes compartidos) para tratar una clase extensa de problemas que se pueden expresar en términos de dichas operaciones.

Los elementos del lenguaje (instancias de procesos, clases y monitores) son declarados y creados mediante sentencias adicionales usando la técnica propuesta de "calificadores". Dichas sentencias adicionales son estructuradas y consistentes con la notación del lenguaje.

Las técnicas propuestas en este trabajo son originales y no dependen del lenguaje ni de la arquitectura del procesador. Aumentan la claridad y expresividad del lenguaje ocultando detalles inecesarios para el programador. Su utilidad se puede medir por la facilidad en el desarrollo de una amplia gama de aplicaciones.

Extender un lenguaje lo acondiciona para ciertas aplicaciones. Permite discernir y jerarquizar la importancia de las operaciones que se proveen, y trata de elevarlas a la categoría de sentencias del lenguaje. Se puede considerar como el prototipo de un lenguaje orientado hacia alguna clase de problemas.

APENDICE I

EXTENSIONES AL LENGUAJE

PL/M-86 CONCURRENTE es una extensión al lenguaje de desarrollo PL/M-86 para tratar problemas que involucran concurrencia. Las extensiones al lenguaje consisten en la definición de nuevos tipos de variables, operaciones y sentencias adicionales. A continuación se describe el uso de estos elementos junto con algunas observaciones importantes para el lector.

TIPOS

Los tipos adicionales se emplean en la misma forma que los proporcionados por el lenguaje. Se definen por macros que son substituidos a tiempo de compilación. Esas definiciones se encuentran en el archivo XINU.EXT.

CONDITION declara que una variable contendrá el identificador de una condición creada por CONDITIONID.

OBJECT declara a una variable como un identificador de objeto asignado por OBJECTID.

PORT declara a una variable como el identificador de un puerto de mensajes estructurados usado para comunicación indirecta entre procesos creado por PCREATE.

SEMAPHORE declara una variable que contendrá el identificador de un semáforo, creado por SCREATE.

OPERACIONES

Los procedimientos constituyen los servicios del núcleo de concurrencia al programa y se podrían clasificar en manejo de memoria, objetos, procesos, comunicación y sincronización. Las declaraciones de estos procedimientos deben incluirse en los módulos del programa. Las declaraciones de las clases se encuentran en el archivo CLASS y las del programa principal en XINU.EXT.

CHPRIO cambia la prioridad de un proceso.

```
chprio (pid, newprio) integer
pid, newprio integer
```

CHPRIO cambia la prioridad para la administración del proceso PID. Las prioridades son enteros positivos. En cualquier momento, el proceso con la más alta prioridad que esté listo para ejecución será el proceso siguiente en ejecución. Un conjunto de procesos con igual prioridad, serán administrados con el método "Round-Robin".

Si la nueva prioridad es inválida, o el identificador de proceso es inválido se produce la condición de error 015H.

Observaciones: Debido a que CHPRIO cambia prioridades sin rearreglar los procesos en la lista de preparados para ejecución, debe usarse sólo sobre procesos en espera, dormidos, suspendidos o en ejecución.

Revisar: CREATE, GETPRIO, RESUME.

CREATE crea un nuevo proceso.

```
create (caddr, ssize, prio, name, nargs, args) integer
     caddr, name, args pointer
     ssize, prio, nargs integer
```

CREATE crea un nuevo proceso que iniciará su ejecución a partir de la localidad CADDR, con una pila de ejecución de SSIZE palabras, prioridad inicial PRIO, y con nombre NAME. CADDR es la dirección de un procedimiento

o programa principal. Si la creación es exitosa, el identificador (no negativo) del nuevo proceso es devuelto al llamador. De otra forma produce la condición de error 013H. El proceso creado permanece en estado suspendido; no iniciará su ejecución hasta que lo active el procedimiento RESUME. Si los argumentos son incorrectos o si no hay espacio en la tabla de procesos, se produce una condición de error. El nuevo proceso tiene su propia pila de ejecución pero comparte datos gobales con otros procesos.

El llamador puede pasar varios argumentos al proceso creado los cuales se accesa a través de los parámetros formales. El entero NARGS especifica el tamaño en bytes que ocupan los argumentos actuales y que deberán tomarse de la dirección indicada por ARGS. Cada argumento se toma como una sola palabra.

CEMPTY pregunta sobre el estado de una condición.

```
cempty (cond) byte
cond condition
```

CEMPTY devuelve TRUE cuando en la condición no hay ningún proceso bloqueado. Da el error 097H cuando se le aplica un identificador de condición inválido.

Revisar: CONDITIONID, CSIGNAL, CWAIT, DO ASSIGN, DO EXCLUSION.

CONDITIONID devuelve un identificador para una variable condición en un monitor.

```
conditionid condition
```

CONDITIONID crea una variable condición usada en un monitor. Produce la condición de error 094H en caso de que la tabla de condiciones esté llena.

Revisar: CEMPTY, CSIGNAL, CWAIT, OBJECTID, DO ASSIGN, DO EXCLUSION.

CSIGNAL reactiva un proceso bloqueado dentro de un monitor.

```
csignal (cond)
cond condition
```

CSIGNAL se emplea solamente dentro de un monitor. Si el identificador de la condición es inválido, produce el error 096H.

Sólo el proceso activo dentro del monitor puede emitir CSIGNAL a una condición. Entonces, cuando salga del monitor, el primer proceso en espera sobre esa condición será activado.

Revisar: CONDITIONID, CEMPTY, CWAIT, DO ASSIGN, DO EXCLUSION.

CWAIT bloquea al proceso que se encuentra dentro del monitor sobre una condición.

cwait (cond)
cond condition

CWAIT se emplea solamente dentro de un monitor. Si el identificador de la condición es inválido, produce el error 095H.

CWAIT causa que el proceso se bloquee y sea reasignado a una cola de procesos en espera sobre la condición. Después, cualquier proceso podrá entrar al monitor.

Revisar: CEMPTY, CONDITIONID, CSIGNAL, DO ASSIGN, DO EXCLUSION.

GETMEM, GETSTACK gestiona un bloque de memoria.

getmem (nbytes) pointer
nbytes integer

Es necesario aclarar que este procedimiento sólo gestiona bloques de memoria a partir de un espacio llamado el "area del sistema", representado por un arreglo de 8 Kbytes de extensión, aun cuando dicho espacio puede ser asignado por el sistema operativo DOS.

GETMEM devuelve la dirección de memoria más baja del bloque asignado; GETSTACK regresa la dirección más alta del bloque asignado. Si hay menos bytes disponibles en el area del sistema, se produce una condición de error.

Las direcciones de memoria obtenidas con GETMEM están normalizadas es decir, la parte desplazamiento del apuntador es cero. El apuntador al bloque obtenido por GETSTACK tiene una parte desplazamiento igual al tamaño del bloque. El tamaño del bloque de memoria obtenido se redondea a los siguientes 16 bytes, si es necesario.

Observaciones: puesto que el 8088 no proporciona mecanismos de protección de memoria, es posible que durante la ejecución del proceso su pila se traslape con otras areas de memoria.

GETPID devuelve el identificador del proceso en ejecución.

getpid integer

GETPID devuelve el identificador del proceso en ejecución. Es necesario poderse identificar él mismo para realizar algunas operaciones (P.ej. cambiar la prioridad con la cual se está administrando).

GETPRIO devuelve la prioridad de un proceso dado.

getprio (pid) integer
pid integer

GETPRIO devuelve la prioridad del proceso PID. Si PID es inválido, se produce una condición de error.

Revisar: GETPID.

KILL termina un proceso.

kill (pid)
pid integer

KILL detendrá al proceso PID y lo remueve del sistema. Si el identificador del proceso es inválido, se produce la condición de error 012H. KILL destruye a cualquier proceso inmediatamente, aun estando en ejecución. Si el proceso está esperando sobre un semáforo, se remueve de la lista de espera y la cuenta del semáforo se incrementa como si el proceso nunca hubiera estado allí. El proceso que estaba esperando enviar un mensaje a un puerto desaparece sin afectar al puerto. Es posible aniquilar a un proceso que se encuentre en cualquier estado, incluyendo el estado suspendido. Una vez destruido el proceso no se puede recuperar.

Observaciones: en este trabajo, no es posible recuperar el espacio de memoria asignado al código del proceso que termina. Sin embargo, KILL recupera el espacio de la pila de ejecución asignado al proceso cuando se crea.

OBJECTID crea un nuevo identificador de objeto.

objectid object

OBJECTID devuelve el identificador de un objeto. Este objeto no pertenece a ninguna clase hasta que no sea relacionado mediante el método de asignación (aquel que contiene la sentencia DO ASSIGN). Este identificador debe aparecer siempre como último argumento en la lista de parámetros de los métodos de la clase.

Produce la condición de error 091H cuando no puede crear mas identificadores de objeto.

Revisar: DO ASSIGN, DO EXCLUSION.

PCOUNT devuelve el número de mensajes que esperan en un puerto.

```
pcount (portid) integer
     portid integer
```

PCOUNT devuelve el número de mensajes asociados al puerto PORTID. Una cuenta positiva P significa que hay P mensajes disponibles para procesamiento. Esta cuenta incluye los mensajes en el puerto y la cuenta de los procesos que intentaron enviar mensajes. Una cuenta negativa -P, significa que hay P procesos esperando mensajes del puerto. Si la cuenta es cero significa que no hay mensajes esperando ni procesos esperando consumir mensajes.

Revisar: PCREATE, PDELETE, PRECEIVE, PSEND, PREAD, PWRITE.

PCONTROL administra el tráfico de mensajes de un grupo de procesos.

```
pcontrol (portid)
     portid integer
```

PCONTROL es invocado por el proceso del sistema PCONTROLLER. Se encarga de reactivar los procesos lectores (PREAD) y escritores (PWRITE) que permanecen bloqueados en un ciclo de intercambio de mensajes compartidos. Posee la prioridad más baja que cualquier proceso del programa, con lo cual se activa cada vez que se bloquean aquellos sobre un puerto. Termina cuando no hay mas procesos que envíen mensajes al puerto.

Revisar: PCREATE, PREAD, PWRITE.

PCREATE crea un puerto nuevo.

```
pcreate (count, msize) integer
     count integer
     msize word
```

PCREATE crea un puerto con espacio para alojar las copias de COUNT mensajes de tamaño MSIZE. Los mensajes pueden ser de cualquier tipo, simple o estructurado, y sólo de ese tipo para un puerto. La función SIZEOF devuelve el tamaño en bytes que ocupa una variable.

PCREATE regresa un entero que identifica al puerto si se realizó exitosamente. Si no hay mas puertos que puedan asignarse, o si la cuenta es no positiva, se produce la condición de error 051H.

Los puertos se pueden manipular con las operaciones PSEND y PRECEIVE que se refieren a los mecanismos usuales de comunicación, y con PWRITE y PREAD que son las operaciones sobre los mensajes compartidos. Sin embargo, no se deben mezclar al aplicarse a un puerto.

Revisar: PCOUNT, PDELETE, PRECEIVE, PSEND, PREAD, PWRITE.

PDELETE destruye un puerto.

```
pdelete (portid)
     portid integer
```

PDELETE libera un puerto. Produce la condición de error 058H, si PORTID es ilegal o no ha sido asignado.

Devuelve la memoria asignada para alojar los mensajes y reactiva los procesos que esperaban enviar o recibir un mensaje.

Revisar: PCOUNT, PCREATE, PRECEIVE, PSEND, PREAD, PWRITE.

PREAD obtiene un mensaje compartido del puerto.

```
pread (portid, buffer)
     portid integer
     buffer pointer
```

PREAD obtiene una copia del mensaje para el proceso que lo llama obtenida del puerto PORTID y depositada en la dirección BUFFER. Si PORTID

es un identificador inválido del puerto o si no ha sido asignado, se produce la condición de error 057H.

PREAD causa que el proceso que lo emite se bloquee siempre. Si cualquier otro proceso intenta obtener un mensaje del mismo puerto, ambos procesos compartirán el mensaje, una vez que el PCONTROL los reactive. Esta operación sólo debe combinarse con PWRITE para enviar mensajes.

Revisar: PCOUNT, PCREATE, PDELETE, PSEND, PWRITE.

PRECEIVE obtiene un mensaje del puerto.

```
preceive (portid, buffer)
     portid integer
     buffer pointer
```

PRECEIVE obtiene un mensaje del puerto PORTID, copiándolo a la dirección del BUFFER. La condición de error 055H se produce cuando PORTID es un identificador ilegal o no haya sido asignado.

El proceso llamador se bloquea si no hay mensajes disponibles. La única forma de que un proceso se libere de un puerto sobre el cual espera es que algún otro proceso envíe un mensaje al puerto con PSEND o bien, otro proceso destruya o restaure el puerto con PDELETE.

Revisar: PCOUNT, PCREATE, PDELETE, PSEND, PREAD, PWRITE.

PSEND envía un mensaje a un puerto.

```
psend (portid, buffer)
     portid integer
     buffer pointer
```

PSEND envía la copia de un mensaje que se encuentra en la dirección BUFFER al puerto PORTID. Si PORTID es un identificador inválido o si el puerto está libre se produce la condición de error 054H.

Si el puerto está lleno en el momento de la llamada, los procesos emisores se bloquearán hasta que haya espacio disponible en el puerto para el mensaje.

Revisar: PCOUNT, PCREATE, PDELETE, PRECEIVE, PREAD, PWRITE.

PWRITE envía un mensaje que será compartido.

```
pwrite (portid, buffer)
     portid integer
     buffer pointer
```

PWRITE envía una copia del mensaje en BUFFER al puerto PORTID para que los procesos que emitan PREAD lo puedan compartir. Un mal identificador PORTID o un puerto no asignado, produce la condición de error 056H.

Si hay espacio disponible en el puerto PWRITE regresa el control al llamador. De otra forma, espera hasta que el administrador del puerto PCONTROL lo reactive. Esta operación sólo debe usarse en combinación de PREAD.

Revisar: PCOUNT, PCREATE, PDELETE, PRECEIVE, PREAD, PSEND.

RECEIVE recibe un mensaje (una palabra).
RECVCLR recibe o quita mensaje.

```
receive integer
recvclr integer
```

RECEIVE regresa mensajes de una palabra enviados por un proceso usando send. Si no había mensajes esperando, RECEIVE se bloquea hasta que aparezca uno.

RECVCLR limpia el espacio para que arriben mensajes, regresando cualquier mensaje que estuviera esperando u "ok" si no había. A diferencia de RECEIVE, RECVCLR nunca se bloquea.

Revisar: SEND, SIGNAL, WAIT

RESUME reanuda un proceso suspendido.

```
resume (pid)
     pid integer
```

RESUME toma un identificador de proceso PID y lo activa, permitiendo reanudar su ejecución. Si PID es inválido, o el proceso PID no está suspendido, se produce la condición de error 010H. Sólo los procesos suspendidos se pueden reanudar.

Revisar: SLEEP, SUSPEND, SEND, RECEIVE.

SCOUNT regresa la cuenta asociada con un semáforo.

scount (sem) integer
    sem integer

SCOUNT devuelve la cuenta actual asociada con el semáforo SEM. Una cuenta negativa -P significa que hay P procesos esperando en el semáforo; una cuenta positiva P significa que a lo más P llamadas a WAIT pueden ocurrir antes de que el proceso se bloquee (suponiendo que no intervino ningún llamado a SIGNAL). Si SEM es un mal identificador de semáforo o no ha sido asignado, se produce la condición de error 027H.

Revisar: SCREATE, SDELETE, SIGNAL, SRESET, WAIT.

SCREATE crea un semáforo nuevo.

screate (count) integer
    count integer

SCREATE crea un nuevo semáforo e inicializa su cuenta a COUNT. SCREATE devuelve el entero identificador del semáforo si fuė llevado a cabo exitosamente, o la condición de error 022H, si no hay mas semáforos disponibles o la cuenta COUNT es negativa.

Los semáforos se manejan con WAIT y SIGNAL para sincronizar procesos. Llamar a WAIT causa que la cuenta del semáforo sea decrementada; decrementar un proceso con una cuenta de cero causa que el proceso se bloquee. Llamar a SIGNAL incrementa la cuenta del semáforo, liberando procesos bloqueados si habia alguno esperando.

Revisar: SCOUNT, SDELETE, SIGNAL, SRESET, WAIT.

SDELETE cancela un semáforo.

sdelete (sem)
    sem integer

SDELETE remueve el semáforo SEM del sistema y reactiva los procesos que estaban esperando sobre este semáforo. La condición de error 024H se produce cuando SEM es un identificador inválido o no ha sido asignado.

Revisar: SCOUNT, SCREATE, SIGNAL, SRESET, WAIT.

SEND envía un mensaje (de una palabra) a un proceso.

```
send (pid, msg)
     pid, msg integer
```

SEND envía un mensaje MSG de una palabra de longitud al proceso con identificador PID. Un proceso puede tener a lo mas un mensaje pendiente que no ha sido recibido.

Si PID es inválido, si el proceso ya tiene un mensaje esperando o si el proceso destino es el mismo que lo emite, se produce la condición de error 030H.

Revisar: RECEIVE, SIGNAL, WAIT.

SIGNAL, SIGNALN envía una señal a un semáforo.

```
signal (sem)
signaln (sem, count)
     sem, count integer
```

SIGNAL envía una señal al semáforo SEM y produce el error 021H, si el semáforo no existe. La llamada a signal incrementa la cuenta de SEM por 1 y libera al siguiente proceso si había alguno esperando. La llamada a SIGNALN incrementa el semáforo por COUNT y libera a COUNT procesos si había tantos esperando. Observe que SIGNALN(SEM, COUNT) es equivalente a ejecutar SIGNAL(SEM), COUNT veces.

Revisar: SCOUNT, SCREATE, SDELETE, SRESET, WAIT

SLEEP, SLEEP10 pone el proceso a dormir por el tiempo especificado.

```
sleep (secs)
sleep10 (tenths)
     secs, tenths integer
```

SLEEP causa que el proceso en ejecución espere por el tiempo especificado y entonces continúe. La forma SLEEP espera por el retardo especificado en segundos; es útil para retardos grandes. La forma SLEEP10 espera el retardo indicado en décimas de segundo, empleada en retardos cortos.

Se produce la condición de error 040H para SLEEP10 o 041H para SLEEP, si el argumento es negativo o si no está habilitado el reloj del sistema.

Hacer que un proceso duerma no es lo mismo que suspenderlo. Entre

otras cosas, un proceso dormido no puede despertarse antes de transcurrido el tiempo.

Revisar: SUSPEND.

Observaciones: el tiempo máximo para dormir es 32767 segundos (cerca de 546 minutos). SLEEP garantiza una cota inferior de retardo, pero puesto que el sistema puede retardar el procesamiento de interrupciones a veces, sleep no puede garantizar una cota superior.

SRESET restaura la cuenta de un semáforo.

```
sreset (sem, count)
     sem, count integer
```

SRESET libera procesos que se encuentren esperando en el semáforo sem, y restablece la cuenta a count. Esto corresponde a la operación de SDELETE(SEM) y SEM = SCREATE(COUNT), excepto que se garantiza que el identificador del semáforo permanece sin cambio. SRESET regresa "syserr" si sem no es un identificador de semáforo válido. La cuenta corriente en un semáforo no afecta el restaurarlo.

Revisar: SCOUNT, SCREATE, SDELETE, SIGNAL, WAIT

STARTDEFCLK inicia el procesamiento del reloj en modo diferido.

STOPDEFCLK detiene el modo diferido y reinicia la operación normal del reloj.

SUSPEND suspende un proceso.

```
suspend (pid)
     pid integer
```

SUSPEND coloca al proceso con identificador PID en estado suspendido Si PID es inválido, o el proceso no está actualmente en ejecución, o no está en la lista de preparados, SUSPEND produce la condición de error 011H.

Observe que la suspensión difiere de dormir porque un proceso suspendido puede permanecer en las colas de entrada/salida o de semáforos. Un proceso puede poner a otro en estado suspendido. Un proceso sólo puede ponerse el mismo a dormir.

Revisar: RESUME, SLEEP, SEND, RECEIVE.

WAIT bloquea y espera hasta que a un semáforo se le mande una señal.

```
wait (sem)
     sem integer
```

WAIT decrementa la cuenta del semáforo SEM, bloqueando al proceso invocador si la cuenta se hace negativa y procede a encolarlo en el semáforo. La única forma para liberar a un proceso de la cola del semáforo es que algún otro proceso señale al semáforo, o que lo elimine, o que restableza su cuenta. WAIT y SIGNAL son las dos primitivas básicas del sistema.

WAIT da la condición de error 020H si SEM es inválido.

Revisar: SCOUNT, SCREATE, SDELETE, SIGNAL, SRESET.

SENTENCIAS

Las sentencias adicionales declaran tipos especiales de construcciones o elementos en el lenguaje. Utilizan el método de los calificadores cuyas llamadas se ocultan por macrodefiniciones para crear procesos, clases o monitores, por lo cual, no se verifica en la compilación que se cumplan en cada caso las reglas que se mencionan a continuación, debiendo cuidar su escritura.

DO ASSIGN crea dinámicamente una nueva instancia de la clase.

```
do assign;
     <SENTENCIAS>
enda;
```

DO ASSIGN crea a tiempo de ejecución una nueva instancia de una clase y le asigna un identificador de objeto. Debe ser la única sentencia más externa en la definición del método de asignación.

La clase debe compilarse separadamente con la opción $LARGE, en un módulo externo al programa principal y unido en la fase de encadenamiento. Los métodos de la clase deben poseer el atributo ENTRY. El identificador del objeto es el único argumento en el método de asignación. Este debe ser el primer método que se aplique a un objeto porque indica su pertencia a una clase. Las definiciones de la clase se encuentran en el archivo CLASS.

Revisar: DO EXCLUSION, OBJECTID.

DO CONCURRENT crea el ambiente de concurrencia.

```
do concurrent;
     <SENTENCIAS>
endc;
```

Inicia las estructuras de datos del núcleo de concurrencia, el sistema de entrada y salida, y reanuda la actividad de los procesos creados y

suspendidos por CREATE o DO PROCESS. En realidad, DO CONCURRENT no realiza la ejecución concurrente de las sentencias que contiene. Su propósito es la de hacer transparente al programador la activación de los procesos creados pero suspendidos.

Esta sentencia debe ser la única y la más externa en el módulo del programa principal. La definición de esta sentencia se encuentra en el archivo XINU.EXT.

Revisar: DO PROCESS, CREATE, KILL, RESUME, SUSPEND.

DO EXCLUSION establece que un procedimiento es un método de la clase.

```
do exclusion;
    <SENTENCIAS>
endx;
```

DO EXCLUSION aplica las operaciones de la definición del método sobre un objeto. PL/M-86 CONCURRENTE asegura una ejecución mutuamente exclusiva entre los métodos para accesar los datos declarados en la clase: sólo un método a la vez puede accesar las variables.

Un monitor es una clase que usa variables de tipo CONDITION, por lo cual, deben usarse en la misma forma. Pueden haber varios procesos dentro de un monitor, pero sólo hay uno activo. Cuando un proceso abandona el monitor, otro puede entrar o ser reactivado, si ya estaba dentro.

La definición de DO EXCLUSION se encuentra en el archivo CLASS.

Revisar: DO ASSIGN, CEMPTY, CSIGNAL, CWAIT, CONDITIONID, OBJECTID.

DO PROCESS establece que un procedimiento será convertido en un proceso.

```
do process;
    <SENTENCIAS>
endp;
```

DO PROCESS convierte un procedimiento en un proceso suspendido. La invocación del procedimiento debe aparecer en el programa principal dentro de la sentencia DO CONCURRENT. El proceso creado tiene una pila de ejecución de tamaño preestablecido y su definición es la misma que la del procedimiento. El proceso se crea cuando el flujo de control entra a la sentencia DO PROCESS, y se aniquila cuando la abandona.

DO PROCESS es un medio más simple y elegante de crear procesos que CREATE y son totalmente compatibles. Los procesos poseen una area de datos

y una pila de ejecución exclusivas. La comunicación con un proceso es por medio de los argumentos del procedimiento (en la inicialización) o por medio de mensajes (en el transcurso de la ejecución).

Esta sentencia debe ser la primera y única en la definición del procedimiento. Los procesos deben declararse en el mismo módulo del programa principal. La definición se encuentra en el archivo XINU.EXT.

Revisar: DO CONCURRENT, CREATE, KILL, RESUME, SUSPEND.

APENDICE II

LISTADOS

A continuación se incluyen los listados de los módulos que componen el núcleo del sistema operativo XINU.

Los módulos son los siguientes:

Administrador de memoria: MEM.
Listas de procesos: Q.
Administrador de procesos: PROC.
Semáforos y comunicación directa: SEM.
Comunicación indirecta: PORTS.
Clases y monitores: DATA.
Administrador del reloj: CLOCK.
Iniciación del sistema: SYSINIT.

Cada uno de los módulos se divide en dos archivos. Un archivo contiene exclusivamente declaraciones de variables usadas en el módulo y tiene extensión .DEC. El segundo archivo contiene los procedimientos y tiene la extensión .P86. Los procedimientos que exporta el módulo poseen el atributo PUBLIC.

No se incluyen los archivos con extensión .EXT correspondientes a cada módulo. Tales archivos contienen las declaraciones de los procedimientos que se definen como públicos en dichos módulos. Por ejemplo, la operación WAIT se define en el módulo SEM.P86. Entonces, la siguiente declaración debe aparecer en el archivo SEM.EXT:

```
wait: procedure(sem) external;
     declare sem integer;
end wait;
```

Se deja al lector interesado en este trabajo la tarea de escribir dichos archivos.

```
/* CONF.DEC: DEFINICIONES Y VARIABLES GLOBALES.
*/

$save nolist
declare
   intvector  literally 'OFFH',         /* VECTOR DE INTERRUPCION DE CTXSW */
   reschedule literally 'call resched',/* FORZA CAMBIO DE CONTEXTO        */
   clkvector  literally '1CH',          /* VECTOR DE INTERRUPCION DE CLKINT*/
   isbadpid   literally '(pid < 0 or pid >= nproc)',
   prcurr     literally '01',           /* PROCESO EN EJECUCION           */
   prfree     literally '02',           /* ENTRADA DISPONIBLE (PROCTAB)   */
   prready    literally '03',           /* PROCESO ELEGIBLE               */
   prrecv     literally '04',           /*    "    ESPERANDO RECIBIR      */
   prsleep    literally '05',           /*    "    DURMIENDO              */
   prsusp     literally '06',           /*    "    SUSPENDIDO             */
   prwait     literally '07',           /*    "    ESPERANDO EN SEMAFORO  */
   pnmlen     literally '16',           /* LONG.MAX. DEL NOMBRE DEL PROCESO*/

/* DESPLAZAMIENTO RELATIVO AL REGISTRO BP DURANTE UNA INTERRUPCION        */
   PS         literally '0BH',
   CS         literally '0AH',
   IP         literally '09H',
   AX         literally '08H',
   CX         literally '07H',
   DX         literally '06H',
   BX         literally '05H',
   SI         literally '04H',
   DI         literally '03H',
   ES         literally '02H',
   DS         literally '01H',
   BP         literally '00H',

declare
   nproc      literally '32';           /* NUMERO MAXIMO DE PROCESOS      */

declare
   nil_       literally  '-1',          /* VALOR ESPECIAL                 */
   null       literally  '0',           /* APUNTADOR NULO                 */
   nullproc   literally  '0',           /* IDENTIFICADOR DEL PROCESO NULO */
   reschyes   literally  '1',           /* CAMBIAR CONTEXTO!              */
   reschno    literally  '0',           /* NO CAMBIAR CONTEXTO!           */
   minint     literally  '-32767',      /* MENOR ENTERO POSIBLE           */
   maxint     literally  ' 32767',      /* MAYOR ENTERO POSIBLE           */
   minstk     literally  '255',         /* TAMANO DE LA PILA DE EJECUCION */
   ok         literally  '1',           /* RESULTADO DE OPERACION CORRECTO */
   syserr     literally  'OFFH',        /* RESULTADO INCORRECTO           */
   initprio   literally  '16',          /* PRIORIDAD INICIAL              */
   quantum    literally  '4',           /* TIEMPO PARA LA EJECUCION       */
   dlyclk     literally  '2',           /* RETARDO DEL RELOJ              */
```

```
   semaphore  literally 'integer',    /* DEFINE EL "TIPO" SEMAFORO        */
   port       literally 'integer',    /*         "        PUERTO          */
   class      literally 'integer',    /*         "        CLASE           */
   object     literally 'word',       /*         "        OBJETO          */
   condition  literally 'word',       /*         "        CONDICION       */
   loop       literally 'while true', /* DEFINE UN CICLO INFINITO         */

/* DEFINE LAS SENTENCIAS ADICIONALES OCULTANDO LLAMADAS A PROCEDIMIENTOS */
   process    literally ';call process_; return',          /* DO PROCESS */
   endp       literally 'call kill(currpid); end',         /* ENDP       */
   concurrent literally ';call sysinit; call setcursoroff',/* DO CONCURRENT*/
   endc       literally 'call sysend; end';                /* ENDC       */

declare
   nullpid integer,                   /* IDENTIFICADOR DEL PROCESO NULO  */
   preempt integer;                   /* REASIGNACION DEL PROCESADOR
                                         POR TIEMPO CONSUMIDO            */

$restore
```

```
/* Q.DEC: ESTRUCTURA DE LAS LISTAS DE PROCESOS.
*/

$save nolist

declare
   empty      literally  'OFFH',     /* APUNTADOR NULO               */
   nqent      literally  '0A4H',     /* NPROC + NSEM + NSEM + 4      */
   qentry     literally
   'structure(
      qkey    integer,               /* CLAVE DE ORDEN EN LA LISTA   */
      qnext   integer,               /* INDICE DEL PROCESO SIGUIENTE */
      qprev   integer                /* INDICE DEL PROCESO PREVIO    */
   )';

declare
   q (nqent) qentry ,                /* LISTA DE PROCESOS Y SEMAFOROS*/
   nextqueue integer;                /* ENTRADA SIGUIENTE DISPONIBLE */

$restore

/*   Q.P86 (QUEUE): MANEJO DE LISTAS DE ESPERA DE PROCESOS.

     OPERACIONES: QINIT, ENQUEUE, DEQUEUE, INSERT, INSERTD,
                  GETFIRST, GETLAST, ISEMPTY, FIRSTKEY, LASTKEY,
                  ADDRFIRSTKEY, FIRSTID, NEWQUEUE.
*/

$large optimize(3) save nolist

queue: do;
$include (conf.dec)
$include (q.dec)
$include (sysinit.ext)


/* QINIT: INICIA EL APUNTADOR DE LA PRIMERA ENTRADA DISPONIBLE.
*/
   qinit: procedure public;

      nextqueue = nproc;
   end qinit;


/* ENQUEUE: INSERTA UN ELEMENTO AL FINAL DE LA LISTA.
*/
   enqueue: procedure(index, tail) integer public;
      declare (index, tail) integer;

      q(index).qnext = tail;
```

```
      q(index).qprev = q(tail).qprev;
      q(q(tail).qprev).qnext = index;
      q(tail).qprev  = index;
      return index;
   end enqueue;


/* DEQUEUE: EXTRAE UN ELEMENTO DE LA LISTA Y LO REGRESA.
*/
   dequeue: procedure(index) integer public;
      declare index integer;

      q(q(index).qprev).qnext = q(index).qnext;
      q(q(index).qnext).qprev = q(index).qprev;
      return index;
   end dequeue;


/* INSERT: INSERTA UN PROCESO EN ORDEN DE SU CLAVE.
*/
   insert: procedure(proc, head, prio) public;
      declare (proc, head, prio) integer;
      declare (next, prev) integer;

      next = q(head).qnext;
      do while q(next).qkey < prio;
         next = q(next).qnext;
      end;
      q(proc).qnext = next;
      q(proc).qprev, prev = q(next).qprev;
      q(proc).qkey  = prio;
      q(prev).qnext = proc;
      q(next).qprev = proc;
   end insert;


/* INSERTD: INSERTA UN PROCESO EN LA LISTA DE DORMIDOS.
*/
   insertd: procedure(pid, head, key) public;
      declare (pid, head, key) integer;
      declare (next, prev) integer;

      prev = head;
      next = q(head).qnext;
      do while q(next).qkey < key;
         key  = key - q(next).qkey;
         prev = next;
         next = q(next).qnext;
      end;
      q(pid).qnext  = next;
```

```
      q(pid).qprev  = prev;
      q(pid).qkey   = key;
      q(prev).qnext = pid;
      q(next).qprev = pid;
      if next < nproc
         then q(next).qkey = q(next).qkey - key;
   end insertd;


/* GETFIRST: EXTRAE Y REGRESA EL PRIMER PROCESO EN LA LISTA.
*/
   getfirst: procedure(head) integer public;
      declare head integer;
      declare proc integer;

      if (proc := q(head).qnext) < nproc
         then
            return dequeue(proc);
         else
            return nil_;
   end getfirst;


/* GETLAST: EXTRAE Y REGRESA EL ULTIMO PROCESO EN LA LISTA.
*/
   getlast: procedure(tail) integer public;
      declare tail integer;
      declare proc integer;

      if (proc := q(tail).qprev) < nproc
         then
            return dequeue(proc);
         else
            return nil_;
   end getlast;


/* ISEMPTY: INDICA SI LA LISTA ESTA VACIA.
*/
   isempty: procedure(list) byte public;
      declare list integer;

      return q(list).qnext >= nproc;
   end isempty;


/* FIRSTKEY: REGRESA SIN EXTRAER LA CLAVE DEL PRIMER PROCESO DE LA LISTA.
*/
   firstkey: procedure(list) integer public;
      declare list integer;
```

```
      return q(q(list).qnext).qkey;
   end firstkey;


/* LASTKEY: REGRESA SIN EXTRAER LA CLAVE DEL ULTIMO PROCESO DE LA LISTA
*/
   lastkey: procedure(list) integer public;
      declare list integer;

      return q(q(list).qprev).qkey;
   end lastkey;


/* ADDRFIRSTKEY: REGRESA LA DIRECCION DE LA CLAVE DEL PRIMER PROCESO.
*/
   addrfirstkey: procedure(list) pointer public;
      declare list integer;

      return @q(q(list).qnext).qkey;
   end addrfirstkey;


/* FIRSTID: ES EL IDENTIFICADOR DEL PRIMER PROCESO DE LA LISTA.
*/
   firstid: procedure(list) integer public;
      declare list integer;

      return q(list).qnext;
   end firstid;


/* NEWQUEUE: CREA UNA NUEVA LISTA DENTRO DE LA ESTRUCTURA Q.
*/
   newqueue: procedure integer public;
      declare (head, tail)  integer;

      head = nextqueue;
      nextqueue = nextqueue + 1;
      tail = nextqueue;
      nextqueue = nextqueue + 1;
      q(head).qnext = tail;
      q(head).qprev = nil_;
      q(head).qkey  = minint;
      q(tail).qnext = nil_;
      q(tail).qprev = head;
      q(tail).qkey  = maxint;
      return head;
   end newqueue;
```

```
end queue;
$restore


/* MEM.DEC: ADMINISTRADOR DE MEMORIA DE BAJO NIVEL.
            DECLARA LA ESTRUCTURA DE LOS BLOQUES DE MEMORIA.
*/

declare
   sizefreemem literally '8010H',      /* TAMANO DEL AREA DEL SISTEMA           */
   freememory (sizefreemem) byte,      /* AREA DEL SISTEMA                      */
   mblock literally 'structure(
      next word,                       /* SEGMENTO DEL BLOQUE SIGUIENTE         */
      size word                        /* TAMANO DE ESTE BLOQUE                 */
   )',
   memlist mblock,                     /* DESCRIPTOR DEL PRIMER BLOQUE          */
   maxaddr word,                       /* EXTREMO (SEGMENTO) SUPERIOR DEL AREA*/
   minaddr word;                       /* EXTREMO (SEGMENTO) INFERIOR DEL AREA*/


/* MEM.P86 (MEM): ADMINISTRADOR DE MEMORIA DE BAJO NIVEL.

   OPERACIONES: MEMINIT, GETSTACK, FREESTACK, GETMEM, FREEMEM.
*/

$large optimize(3) save nolist

mem: do;
$include (es.ext)
$include (conf.dec)
$include (mem.dec)
$include (sysinit.ext)


/* MEMINIT: INICIA LA LISTA DE BLOQUES DE MEMORIA.
*/
   meminit: procedure public;
      declare u pointer, u_ based u mblock;

      maxaddr = seg(selectorof(@freememory(sizefreemem - 0FH)))
         + shr((offsetof(@freememory(sizefreemem - 0FH)) + 0FH), 4);
      minaddr = seg(selectorof(@freememory))
         + shr((offsetof(@freememory) + 0FH), 4);
      memlist.next = minaddr;
      memlist.size = sizefreemem - 10H;
      u = buildptr(sel(minaddr),0);
      u_.next = maxaddr;
      u_.size = sizefreemem - 10H;
      u = buildptr(sel(maxaddr),0);
```

```
      u_.next, u_.size = 0;
   end meminit;


/* GETSTACK: ASIGNA MEMORIA PARA LA PILA DE EJECUCION.
             REGRESA LA DIRECCION DEL EXTREMO SUPERIOR DEL BLOQUE.
*/
   getstack: procedure(nbytes) pointer public;
      declare nbytes word;
      declare size word;
      declare (p, q, fits, fitsq) pointer;
      declare (p_ based p, q_ based q, fits_ based fits, fitsq_ based fitsq) mblock

      disable;
      if nbytes <= 0 then
         call sysfail(0H);
      nbytes = (nbytes + 0FH) and 0FFF0H;
      fits = nil;
      q = @memlist;
      p = buildptr(sel(q_.next), 0);
      do while p <> nil;
         if p_.size >= nbytes then
            do;
               fitsq = q;
               fits  = p;
            end;
         q = p;
         p = buildptr(sel(p_.next), 0);
      end;
      if fits = nil then
         call sysfail(1H);
      if nbytes = fits_.size then
         fitsq_.next = fits_.next;
      else
         do;
            fitsq_.next = seg(selectorof(fits)) + shr(nbytes, 4);
            fitsq_.size = 0;
            p = buildptr(sel(fitsq_.next), 0);
            p_.next = fits_.next;
            p_.size = fits_.size - nbytes;
         end;
      enable;
      return buildptr(selectorof(fits), nbytes - 2);
   end getstack;


/* FREESTACK: LIBERA LA MEMORIA ASIGNADA POR GETSTACK.
*/
   freestack: procedure(block) public;
      declare (block, p, q) pointer;
```

```
      declare (block_ based block, p_ based p, q_ based q) mblock;
      declare (size, top, blk) word;

      disable;
      if (size := offsetof(block)) = 0
         or (blk := seg(selectorof(block))) > maxaddr or blk < minaddr
         then
            call sysfail(2H);
      size = (size + 0FH) and 0FFF0H;
      p = buildptr(sel(memlist.next), 0);
      q = @memlist;
      do while p <> nil and p < block;
         q = p;
         p = buildptr(sel(p_.next), 0);
      end;
      if (top := shr(q_.size, 4) + seg(selectorof(q))) > blk and q <> @memlist
         or p <> nil and shr(size, 4) + blk > seg(selectorof(p))
         then
            call sysfail(3H);
      block = buildptr(sel(blk), 0);
      if q <> @memlist and top = blk then
         q_.size = q_.size + size;
      else
         do;
            block_.size = size;
            block_.next = seg(selectorof(p));
            q_.next = blk;
            q = block;
         end;
      if shr(q_.size, 4) + seg(selectorof(q)) = seg(selectorof(p)) then
         do;
            q_.size = q_.size + p_.size;
            q_.next = p_.next;
         end;
      enable;
   end freestack;


/* GETMEM: OBTIENE MEMORIA DEL "HEAP"
           REGRESA LA DIRECCION MAS BAJA DEL BLOQUE.
*/
   getmem: procedure(size) pointer public;
      declare size word;

      return buildptr(selectorof(getstack(size)), 0);
   end getmem;


/* FREEMEM: LIBERA EL BLOQUE DE MEMORIA. LO REGRESA A LA LISTA DE BLOQUES.
*/
```

```
   freemem: procedure(addr, size) public;
      declare addr pointer, size word;

      call freestack(buildptr(selectorof(addr), size));
   end freemem;

end mem;
$restore
```

```
/* PROC.DEC: ADMINISTRADOR DE PROCESOS.
*/

$save nolist

declare
/* DESCRIPTOR DE PROCESOS              */
   pentry     literally 'structure(
      ss       word,                  /* SALVA REGISTRO SEGMENTO DE LA PILA      */
      sp       word,                  /* SALVA REGISTRO TOPE DE LA PILA          */
      psize    word,                  /* TAMANO (BYTES) DE LA PILA DEL PROCESO */
      pstate   integer,               /* ESTADO DEL PROCESO                       */
      pprio    integer,               /* PRIORIDAD DEL PROCESO                    */
      psem     integer,               /* ESPERANDO (WAIT) EN ESTE SEMAFORO        */
      pmsg     integer,               /* MENSAJE RECIBIDO DIRECTAMENTE            */
      phasmsg  integer,               /* NUMERO DE MENSAJES RECIBIDOS             */
      pname (pnmlen) byte             /* NOMBRE DEL PROCESO                       */
   )';

declare
   proctab  (nproc) pentry public,    /* TABLA DE DESCRIPTORES DE PROCESOS       */
   running  byte public,              /* EN EJECUCION CONCURRENTE?               */
   numproc  integer public,           /* NUMERO DE PROCESOS CREADOS              */
   nextproc integer public,           /* SIGUIENTE ENTRADA DISPONIBLE (PROCTAB)*/
   currpid  integer public;           /* IDENTIFICADOR DEL PROCESO EN EJECUCION*/

/* APUNTADORES A LA LISTA DE PROCESOS ELEGIBLES         */
   rdyhead integer,                   /* APUNTADOR AL PRIMER PROCESO             */
   rdytail integer;                   /* APUNTADOR AL ULTIMO PROCESO             */

$restore

$large optimize(3) save nolist

/*   PROC.P86 (PROC) : ADMINISTRADOR DE PROCESOS.

     OPERACIONES: PROCINIT, RESCHED, SUSPEND, RESUME,
                  CREATE, KILL, DOPROCESS, GETPID, GETPRIO, CHPRIO
*/

proc: do;
$include (common.lit)
$include (es.ext)
$include (conf.dec)
$include (proc.dec)
$include (mem.ext)
$include (q.ext)
$include (sem.ext)
$include (sysinit.ext)
```

```
/* PROCINIT: INICIA LA TABLA DE DESCRIPTORES DE PROCESOS. CREA EL PROCESO NULO
             ASIGNA A CTXSW EL VECTOR DE INTERRUPCION OFFH.
*/
   procinit: procedure public;
      declare i integer;

      numproc   = 0;
      nextproc  = nproc - 1;
      do i = 0 to nproc - 1;
         proctab(i).pstate = prfree;
      end;
      rdytail = 1 + (rdyhead := newqueue);
      currpid = nullproc;
      running = false;
      proctab(currpid).pstate = prcurr;
      proctab(currpid).pprio  = 1;
      proctab(currpid).psize  = 0;
      call movb(@('UOI',0), @proctab(currpid).pname, pnmlen);
      call setinterrupt(intvector, ctxsw);
   end procinit;


/* RESCHED: REASIGNA EL PROCESADOR AL PROCESO ELEGIBLE CON MAS ALTA PRIORIDAD.
*/
   resched: procedure public;
      if proctab(currpid).pstate = prcurr
            and lastkey(rdytail) < proctab(currpid).pprio
         then return;
      if proctab(currpid).pstate = prcurr
         then
            do;
               proctab(currpid).pstate = prready;
               call insert(currpid, rdyhead, proctab(currpid).pprio);
            end;
      causeinterrupt(intvector);
   end resched;


/* CTXSW: REALIZA EL CAMBIO DE CONTEXTO. SALVA Y RESTAURA LOS REGISTROS.
*/
   ctxsw: procedure interrupt intvector;
      disable;
      preempt = quantum;
      proctab(currpid).ss = stackbase;
      proctab(currpid).sp = stackptr;
      proctab(currpid := getlast(rdytail)).pstate = prcurr;
      stackbase = proctab(currpid).ss;
      stackptr  = proctab(currpid).sp;
      enable;
```

```
   end ctxsw;


/* READY: HACE ELEGIBLE A UN PROCESO. PUEDE FORZAR QUE ENTRE EN EJECUCION
*/
   ready: procedure(pid, resch) public;
      declare pid integer, resch byte;

      if not (isbadpid)
         then
            do;
               proctab(pid).pstate = prready;
               call insert(pid, rdyhead, proctab(pid).pprio);
               if resch = reschYes
                  then reschedule;
            end;
   end ready;


/* RESUME: CONVIERTE EN ELEGIBLE A UN PROCESO SUSPENDIDO.
           REGRESA SU IDENTIFICADOR.
*/
   resume: procedure(pid) integer public;
      declare pid integer;
      declare prio integer;

      disable;
      if isbadpid or proctab(pid).pstate <> prsusp
         then
            call sysfail(10H);
         else
            do;
               prio = proctab(pid).pprio;
               call ready(pid, reschYes);
            end;
      enable;
      return pid;
   end resume;


/* SUSPEND: SUSPENDE UN PROCESO ELEGIBLE O EN EJECUCION.
            DEVUELVE SU IDENTIFICADOR
*/
   suspend: procedure(pid) integer public;
      declare pid integer;
      declare (prio, v) integer;

      disable;
      if isbadpid or pid = nullproc or (proctab(pid).pstate = prcurr
         and proctab(pid).pstate = prready)
```

```
        then
           call sysfail(11H);
        else
           do;
              if proctab(pid).pstate = prready
                 then
                    do;
                       v = dequeue(pid);
                       proctab(pid).pstate = prsusp;
                    end;
                 else
                    do;
                       proctab(pid).pstate = prsusp;
                       reschedule;
                    end;
              prio = proctab(pid).pprio;
           end;
     enable;
     return prio;
  end suspend;


/* KILL: DESTRUYE UN PROCESO. LIBERA LOS RECURSOS ASIGNADOS A EL.
*/
  kill: procedure(pid) public;
     declare pid integer;

     disable;
     if isbadpid or proctab(pid).pstate = prfree
        then
           call sysfail(12H);
        else
           do;
              declare v integer;

              numproc = numproc - 1;
              call freestack(buildptr(sel(proctab(pid).ss), proctab(pid).psize));
              do case proctab(pid).pstate;
                 ;
              current:
                 do;
                    proctab(pid).pstate = prfree;
                    reschedule;
                 end;
              free:
                 ;
              ready:
                 v = dequeue(pid);
              receive:
                 ;
```

```
            sleep:
               :
            suspend:
               :
            waiting:
               call signal(proctab(pid).psem);
            end;
         end;
      proctab(pid).pstate = prfree;
      enable;
   end kill;


/* CREATE: CREA UN NUEVO PROCESO. REGRESA SU IDENTIFICADOR.
*/
   create: procedure(procaddr, stcksize, priority, name, nargs, args)
      integer public;
      declare (procaddr, stckaddr, name, args) pointer;
      declare stcksize  word;
      declare priority  integer;
      declare nargs     byte;
      declare pid       integer;
      declare stack     based stckaddr (*) word;

      disable;
      if stcksize < minstk or (stckaddr := getstack(stcksize)) = nil
         or (pid := newpid) = syserr or priority < 1
         then
            call sysfail(13H);
         else
            do;
               numproc = numproc + 1;
               proctab(pid).pstate = prsusp;
               proctab(pid).pprio  = priority;
               proctab(pid).psize  = minstk;
               proctab(pid).ss     = seg(selectorof(stckaddr));
               proctab(pid).sp     = offsetof(stckaddr) - shl(double(nargs), 1);
               stckaddr = buildptr(sel(proctab(pid).ss), proctab(pid).sp);
               call movb(name, @proctab(pid).pname, pnmlen);
               call movw(args, stckaddr, nargs);
               proctab(pid).sp     = proctab(pid).sp - 24;
               stckaddr  = buildptr(sel(proctab(pid).ss), proctab(pid).sp);
               stack(PS) = 0F200H;
               stack(CS) = seg(selectorof(procaddr));
               stack(IP) = offsetof(procaddr) + 3;
               stack(BP) = proctab(pid).sp + 20;
               stack(DS) = proctab(pid).ss;
            end;
      enable;
      return pid;
```

```
   end create;


/* DOPROCESS (CALIFICADOR): CREA UN PROCESO.
   SENTENCIA ADICIONAL: DO PROCESS;...ENDP;.
*/
   doprocess: procedure interrupt 0 public;
      declare pstack_ pointer, pstack based pstack_ (*) word;
      declare cstack_ pointer, cstack based cstack_ (*) word;
      declare (a, b, c) wordi, pid integer;

      if (pstack_ := getstack(minstk)) = nil or (pid := newpid) = syserr
         then
            call sysfail(13H);
         else
            do;
               cstack_ = buildptr(sel(stackbase), 0);
               c = shr(cstack(b := shr(cstack(a := shr(stackptr, 1)), 1)), 1);
               numproc = numproc + 1;
               proctab(pid).pstate = prready;
               proctab(pid).pprio  = initprio;
               proctab(pid).psize  = minstk;
               proctab(pid).ss     = seg(selectorof(pstack_));
               proctab(pid).sp     = offsetof(@pstack(minstk - c + a));
               call movw(@cstack(a), @pstack(minstk - c + a), (c - a));
               pstack(minstk - c + a + BP) = offsetof(@pstack(minstk - c + b));
               pstack(minstk - c + a + IP) = pstack(minstk - c + a + IP) + 2
                  + double(-shl(c - b > 2, 1));
               call insert(pid, rdyhead, initprio);
            end;
   end doprocess;


/* NEWPID: REGRESA UN IDENTIFICADOR DE PROCESO DISPONIBLE.
*/
   newpid: procedure integer;
      declare (i, pid) integer;

      do i = 0 to nproc - 1;
         pid = nextproc;
         nextproc = nextproc - 1;
         if pid <= 0 then
            nextproc = nproc - 1;
         if proctab(pid).pstate = prfree then
            return pid;
      end;
      call sysfail(14H);
   end newpid;
```

```
/* CHPRIO: CAMBIA LA PRIORIDAD DE UN PROCESO.
*/
   chprio: procedure(pid, newprio) public;
      declare (pid, newprio) integer;

      disable;
      if isbadpid or newprio <= 0 or proctab(pid).pstate = prfree
         then
            call sysfail(15H);
         else
            proctab(pid).pprio = newprio;
      enable;
   end chprio;


/* GETPRIO: REGRESA LA PRIORIDAD DEL PROCESO.
*/
   getprio: procedure(pid) integer public;
      declare pid integer;

      if isbadpid or proctab(pid).pstate = prfree
         then
            call sysfail(16H);
         else
            return proctab(pid).pprio;
   end getprio;


/* GETPID: DEVUELVE EL IDENTIFICADOR DEL PROCESO EN EJECUCION.
*/
   getpid: procedure integer public;

      return currpid;
   end getpid;

end proc;
$restore
```

```
/* SEM.DEC: SINCRONIZACION DE PROCESOS POR SEMAFOROS.
*/

declare
   nsem     literally  '40H',          /* NUMERO MAXIMO DE SEMAFOROS      */
   isbadsem literally '(sem < 0 or sem >= nsem)',
   sentry   literally  'structure(
      sstate byte,                     /* SEMAFORO DISPONIBLE?            */
      scount integer,                  /* CONTADOR DE ESTE SEMAFORO       */
      sqhead integer,                  /* INDICE DEL PRIMERO DE LA LISTA  */
      sqtail integer                   /* INDICE DEL ULTIMO DE LA LISTA   */
   )';

declare
   semaph  (nsem) sentry,              /* TABLA DE SEMAFOROS              */
   nextsem integer;                    /* SIGUIENTE SEMAFORO DISPONIBLE   */


/* SEM.P86 (SEMAPHORE): SINCRONIZACION DE PROCESOS CON SEMAFOROS.

   PROCESOS: SEMINIT, WAIT, SIGNAL, SCREATE, SDELETE,
             SCOUNT, SIGNALN, SRESET, SLIST.
*/

$large optimize(3) save nolist

semaphore: do;
$include (common.lit)
$include (conf.dec)
$include (sem.dec)
$include (q.ext)
$include (proc.ext)
$include (sysinit.ext)


/* SEMINIT: INICIA LA TABLA DE SEMAFOROS.
*/
   seminit: procedure public;
      declare i integer;

      nextsem = nsem - 1;
      do i = 0 to nsem - 1;
         semaph(i).sstate = sfree;
         semaph(i).sqtail = 1 + (semaph(i).sqhead := newqueue);
      end;
   end seminit;


/* WAIT: CAUSA QUE EL PROCESO EN EJECUCION ESPERE EN EL SEMAFORO.
*/
```

```
   wait: procedure(sem) public;
      declare sem integer;
      declare v integer;

      disable;
      if isbadsem or semaph(sem).sstate = sfree
         then
            call sysfail(20H);
         else
            if (semaph(sem).scount := semaph(sem).scount - 1) < 0
               then
                  do;
                     proctab(currpid).pstate = prwait;
                     proctab(currpid).psem   = sem;
                     v = enqueue(currpid, semaph(sem).sqtail);
                     reschedule;
                  end;
      enable;
   end wait;


/* SIGNAL: SENALA AL SEMAFORO PARA LIBERAR UN PROCESO EN ESPERA.
*/
   signal: procedure(sem) public;
      declare sem integer;
      declare s integer;

      disable;
      if isbadsem or semaph(sem).sstate = sfree
         then
            call sysfail(21H);
         else
            do;
               semaph(sem).scount = (s := semaph(sem).scount) + 1;
               if s < 0 then
                  call ready(getfirst(semaph(sem).sqhead), reschyes);
            end;
      enable;
   end signal;


/* SCREATE: CREA E INICIA UN SEMAFORO. REGRESA SU IDENTIFICADOR.
*/
   screate: procedure(count) integer public;
      declare count integer;
      declare sem   integer;

      disable;
      if count < 0 or (sem := newsem) = syserr
         then
```

```
            call sysfail(22H);
         else
            semaph(sem).scount = count;
      enable;
      return sem;
   end screate;


/* NEWSEM: OBTIENE UN IDENTIFICADOR DE SEMAFORO DISPONIBLE.
*/
   newsem: procedure integer;
      declare sem integer;
      declare i   integer;

      do i = 0 to nsem - 1;
         sem = nextsem;
         nextsem = nextsem - 1;
         if nextsem < 0 then
            nextsem = nextsem - 1;
         if semaph(sem).sstate = sfree then
            do;
               semaph(sem).sstate = sUsed;
               return sem;
            end;
      end;
      call sysfail(23H);
   end newsem;


/* SDELETE: DESTRUYE UN SEMAFORO. LIBERA TODOS LOS PROCESOS QUE ESPERAN EN EL
*/
   sdelete: procedure(sem) public;
      declare sem integer;
      declare p integer;

      disable;
      if isbadsem or (semaph(sem).sstate = sfree)
         then
            call sysfail(24H);
         else
            do;
               semaph(sem).sstate = sfree;
               if not isempty(semaph(sem).sqhead) then
                  do;
                     do while ((p := getfirst(semaph(sem).sqhead)) >= 0)
                           and (p < nproc);
                        call ready(p, false);
                     end;
                     reschedule;
                  end;
```

```
            end;
        enable;
    end sdelete;


/* SRESET: RESTABLECE EL CONTADOR DE UN SEMAFORO A UN VALOR VALIDO.
*/
    sreset: procedure(sem, n) public;
        declare sem integer, n integer;

        disable;
        if isbadsem or (semaph(sem).sstate = sfree) or (n < 0)
            then
                call sysfail(25H);
            else
                do;
                    do while semaph(sem).scount < 0;
                        call ready(getfirst(semaph(sem).sqhead), false);
                        semaph(sem).scount = semaph(sem).scount + 1;
                    end;
                    semaph(sem).scount = n;
                    reschedule;
                end;
        enable;
    end sreset;


/* SIGNALN: SENALA AL SEMAFORO N VECES.
*/
    signaln: procedure(sem, n) public;
        declare (sem, n) integer;

        disable;
        if isbadsem or (semaph(sem).sstate = sfree) or (n < 0)
            then
                call sysfail(26H);
            else
                do;
                    do while n > 0;
                        if semaph(sem).scount < 0 then
                            call ready(getfirst(semaph(sem).sqhead), false);
                        semaph(sem).scount = semaph(sem).scount + 1;
                        n = n - 1;
                    end;
                    reschedule;
                end;
        enable;
    end signaln,
```

```
/* SCOUNT: REGRESA LA CUENTA DEL SEMAFORO.
*/
   scount: procedure(sem) integer public;
      declare sem integer;

      if isbadsem or semaph(sem).sstate = sfree
         then
            call sysfail(27H);
         else
            return semaph(sem).scount;
   end scount;


/* SLIST: REGRESA EL PRIMER ELEMENTO DE LA LISTA
*/
   slist: procedure(sem) integer public;
      declare sem integer;

      if isbadsem or semaph(sem).sstate = sfree
         then
            call sysfail(28H);
         else
            return semaph(sem).sqhead;
   end slist;

end semaphore;
$restore
```

```
$save nolist

/* CLOCK.DEC: ADMINISTRADOR DEL RELOJ.
*/

declare
   clkruns   byte public,             /* RELOJ FUNCIONANDO?                */
   clockq    integer,                 /* APUNTADOR A LA LISTA DELTAQ       */
   slnempty  byte public,             /* LISTA DELTAQ VACIA?               */
   sltop_    pointer,                 /* APUNTADOR A LA CLAVE DEL PRIMER...*/
   sltop     based sltop_ integer,    /* ... PROCESO EN DELTAQ             */
   defclk    byte,                    /* RELOJ EN MODO DIFERIDO?           */
   delay     integer,                 /* RETARDO PRODUCIDO EN EL RELOJ     */
   slnext    integer;                 /* ELEMENTO SIGUIENTE EN DELTAQ      */

$restore

/* CLOCK (CLOCK): ADMINISTRADOR DEL RELOJ.
     CLKINIT, SLEEP10, SLEEP, CLKINT, STARTDEFCLK, STOPDEFCLK.
*/

$large optimize(3) nointvector save nolist

clock: do;
$include (common.lit)
$include (es.ext)
$include (conf.dec)
$include (mem.ext)
$include (q.ext)
$include (proc.ext)
$include (sem.ext)
$include (sysinit.ext)
$include (clock.dec)


/* CLKINIT: INICIA LA LISTA DELTAQ. ASIGNA LA RUTINA DE SERVICIO AL VECTOR
            DE INTERRUPCION CLKVECTOR (1CH).
*/
   clkinit: procedure public;

      clockq   = newqueue;
      slnext   = clockq;
      clkruns  = false;
      slnempty = false;
      defclk   = false;
      delay    = dlyclk;
      call setinterrupt(clkvector, clkint);
   end clkinit;
```

```
/* SLEEP10: DUERME AL PROCESO EN EJECUCION POR EL TIEMPO ESPECIFICADO
            EN DECIMAS DE SEGUNDO (APROXIMADAMENTE).
*/
   sleep10: procedure(n) public;
      declare n integer;

      disable;
      if n <= 0 or not clkruns
         then
            call sysfail(40H);
         else
            do;
               call insertd(currpid, clockq, n);
               slnempty = true;
               sltop_    = addrfirstkey(clockq);
               proctab(currpid).pstate = prsleep;
               reschedule;
            end;
      enable;
   end sleep10;


/* SLEEP: DUERME AL PROCESO EN EJECUCION POR HORAS.
*/
   sleep: procedure(n) public;
      declare n integer;

      if n <= 0 or not clkruns
         then
            call sysfail(41H);
         else
            do;
               do while n >= 1000;
                  call sleep10(10000);
                  n = n - 1000;
               end;
               if n > 0
                  then call sleep10(10 * n);
            end;
   end sleep;


/* WAKEUP: DESPIERTA PROCESOS HACIENDOLOS ELEGIBLES.
*/
   wakeup: procedure;

      do while (not isempty(clockq)) and (firstkey(clockq) <= 0);
         call ready(getfirst(clockq), resc[illegible]o);
      end;
      if slnempty := not isempty(clockq)
```

```
         then sltop_ = addrfirstkey(clockq);
      slnext  = clockq;
      preempt = 0;
   end wakeup;


/* MAKEUP: OBTIENE EL SIGUIENTE PROCESO DE LA LISTA DELTAQ CUANDO EL RELOJ
           OPERA EN MODO DIFERIDO.
*/
   makeup: procedure;

      slnext = firstid(slnext);
      if slnempty := not isempty(slnext)
         then sltop_ = addrfirstkey(slnext);
      preempt = 0;
   end makeup;


/* STRTDEFCLK: PONE EL RELOJ EN MODO DIFERIDO.
*/
   strtdefclk: procedure public;

       defclk = true;
   end strtdefclk;


/* STOPDEFCLK: PONE EL RELOJ EN MODO NORMAL. ACTIVA LOS PROCESOS QUE DEBIERON
               DESPERTAR (SI LOS HAY).
*/
   stopdefclk: procedure public;

      defclk = false;
      call wakeup;
   end stopdefclk;


/* CLKINT: RUTINA DE SERVICIO DEL RELOJ. SE ACTIVA CON CADA PULSO.
*/
   clkint: procedure interrupt clkvector public;

      disable;
      if clkruns and (delay := delay - 1) <= 0
         then
            do;
               delay = dlyclk;
               if slnempty
                  then
                     if (sltop := sltop - 1) <= 0
                        then
                           if defclk
```

```
                                        then
                                           call makeup;
                                        else
                                           call wakeup;
                        if (preempt := preempt - 1) <= 0
                           then reschedule;
                     end;
         output(20H) = 20H;
         enable;
      end clkint;

end clock;
$restore
```

```
/* PORTS.DEC: ADMINISTRADOR DE MENSAJES DE TIPO ESTRUCTURADO.
             CONTIENE MODIFICACIONES A LA ESTRUCTURA DE LOS PUERTOS DE XINU
*/

$save nolist

declare
   nports    literally '16',          /* NUMERO DE PUERTOS                        */
   maxmsgs   literally '20',          /* NUMERO MAXIMO DE MENSAJES POR PUERTO*/
   isbadport literally '(portid < 0 or portid >= nports)',
   maxsize   literally '255',         /* EXTENSION MAXIMA DE UN MENSAJE          */
   pent      literally 'structure (
      used byte,                      /* PUERTO USADO?                           */
      ssem integer,                   /* SINCRONIZA LOS MENSAJES ENVIADOS        */
      rsem integer,                   /* SINCRONIZA LOS MENSAJES RECIBIDOS       */
      coun word,                      /* NUMERO ACTUAL DE MENSAJES               */
      pcnt integer,                   /* NUMERO MAXIMO DE MENSAJES               */
      size word,                      /* TAMANO TOTAL DEL AREA DE MENSAJES       */
      area word,                      /* TAMANO DE UN MENSAJE                    */
      head word,                      /* LUGAR DEL PRIMER MENSAJE A ENVIARSE */
      tail word,                      /* LUGAR DEL PRIMER MENSAJE A RECIBIRSE*/
      bufr pointer                    /* DIRECCION DEL AREA DE MENSAJES          */
   )',

   ports(nports) pent,                /* TABLA DE PUERTOS                        */
   nextp integer;                     /* SIGUIENTE PUERTO DISPONIBLE             */

$restore

/* PORTS.P86 (PORTS): ADMINISTRADOR DE PUERTOS (COMUNICACION INDIRECTA).
                      MODIFICA COMPLETAMENTE EL DISE%O ORIGINAL DE XINU.

   OPERACIONES: PINIT, PCREATE, PDELETE, PCOUNT, PSEND, PRECEIVE,
                PREAD, PWRITE, PCONTROL.
*/
$large optimize(3) save nolist

ports: do;
$include (common.lit)
$include (conf.dec)
$include (ports.dec)
$include (es.ext)
$include (mem.ext)
$include (q.ext)
$include (proc.ext)
$include (sem.ext)
$include (sysinit.ext)


/* PINIT: INICIA LA TABLA DE PUERTOS.
```

```
*/
   pinit: procedure public;
      declare i integer;

      nextp = nports - 1;
      do i = 0 to nports - 1;
         ports(i).used = false;
      end;
   end pinit;


/* PCREATE: CREA UN NUEVO PUERTO. REGRESA SU IDENTIFICADOR.
*/
   pcreate: procedure(count, size) integer public;
      declare count integer, size word;
      declare (i, p) integer;
      declare portp pointer, port based portp pent;

      disable;
      if (count > 0) and (size > 0) then
         do i = 0 to nports - 1;
            p = nextp;
            nextp = p - 1;
            if p <= 0 then
               nextp = nports - 1;
            portp = @ports(p);
            if not port.used then
               do;
                  port.used = true;
                  port.ssem = screate(count - 1);
                  port.rsem = screate(0);
                  port.pcnt = 0;
                  port.coun = unsign(count);
                  port.area = size;
                  port.size = unsign(count) * size;
                  port.head = 0;
                  port.tail = 0;
                  port.bufr = getmem(size);
                  if port.bufr = nil
                     then call sysfail(51H);
               end;
            enable;
            return p;
         end;
      else
         call sysfail(52H);
      enable;
      return syserr;
   end pcreate;
```

```
/* PDELETE: DESTRUYE UN PUERTO. LIBERA LOS RECURSOS ASIGNADOS A EL
*/
   pdelete: procedure(portid) public;
      declare portid integer;
      declare portp pointer, port based portp pent;

      disable;
      portp = @ports(portid);
      if isbadport or port.used
         then
            call sysfail(58H);
         else
            do;
               port.used = false;
               call freemem(port.bufr, port.size);
               call sdelete(port.ssem);
               call sdelete(port.rsem);
            end;
      enable;
   end pdelete;


/* PCOUNT: REGRESA EL NUMERO DE MENSAJES EN EL PUERTO.
*/
   pcount: procedure(portid) integer public;
      declare portid integer;

      return ports(portid).pcnt;
   end pcount;


/* PSEND: ENVIA LA COPIA DE UN MENSAJE AL PUERTO.
*/
   psend: procedure(portid, buffer) public;
      declare portid integer, buffer pointer;
      declare portp pointer, port based portp pent;

      disable;
      portp = @ports(portid);
      if isbadport or not port.used
         then
            call sysfail(54H);
         else
            do;
               declare portbufrp pointer, portbufr based portbufrp (*) byte;

               portbufrp = port.bufr;
               call wait(port.ssem);
               call movb(buffer, @portbufr(port.tail), port.area);
```

```
                port.tail = port.tail + port.area;
                if port.tail >= port.size then port.tail = 0;
                port.pcnt = port.pcnt + 1;
                call signal(port.rsem);
             end;
       enable;
    end psend;


/* PRECEIVE: RECIBE LA COPIA DE UN MENSAJE DEL PUERTO.
*/
    preceive: procedure(portid, buffer) public;
       declare portid integer, buffer pointer;
       declare portp pointer, port based portp pent;

       disable;
       portp = @ports(portid);
       if isbadport or not port.used
          then
             call sysfail(55H);
          else
             do;
                declare portbufrp pointer, portbufr based portbufrp (*) byte;

                portbufrp = port.bufr;
                call wait(port.rsem);
                call movb(@portbufr(port.head), buffer, port.area);
                port.head = port.head + port.area;
                if port.head >= port.size then port.head = 0;
                port.pcnt = port.pcnt - 1;
                call signal(port.ssem);
             end;
       enable;
    end preceive;


/* PWRITE: ENVIA LA COPIA DE UN MENSAJE COMPARTIDO AL PUERTO.
*/
    pwrite: procedure(portid, buffer) public;
       declare portid integer, buffer pointer;
       declare portp pointer, port based portp pent;

       disable;
       portp = @ports(portid);
       if isbadport or not port.used
          then
             call sysfail(56H);
          else
             do;
                declare portbufrp pointer, portbufr based portbufrp (*) byte;
```

```
              portbufrp = port.bufr;
              call wait(port.ssem);
              call movb(buffer, @portbufr(port.tail), port.area);
              port.tail = port.tail + port.area;
              if port.tail > port.size then port.tail = 0;
              port.pcnt = port.pcnt + 1;
           end;
     enable;
  end pwrite;


/* PREAD: RECIBE LA COPIA DE UN MENSAJE COMPARTIDO POR OTROS PROCESOS.
*/
  pread: procedure(portid, buffer) public;
     declare portid integer, buffer pointer;
     declare portp pointer, port based portp pent;

     disable;
     portp = @ports(portid);
     if isbadport or not port.used
        then
           call sysfail(57H);
        else
           do;
              declare portbufrp pointer, portbufr based portbufrp (*) byte;

              portbufrp = port.bufr;
              call wait(port.rsem);
              call movb(@portbufr(port.head), buffer, port.area);
           end;
     enable;
  end pread;


/* PCONTROL: ADMINISTRA EL TRAFICO DE MENSAJES COMPARTIDOS EN EL PUERTO.
*/
  pcontrol: procedure(portid) public;
     declare portid integer;
     declare pid integer, portp pointer, port based portp pent;

     portp = @ports(portid);
     if isbadport or not port.used
        then
           call sysfail(58H);
        else
           do;
              call chprio(getpid, 3);
              do while not isempty(slist(port.ssem));
                 if scount(port.rsem) < 0 then
```

```
                    do while port.pcnt > 0;
                       call sreset(port.rsem, 0);
                       port.head = port.head + port.area;
                       if port.head > port.size then
                          port.head = 0;
                       port.pcnt = port.pcnt - 1;
                    end;
                 if scount(port.ssem) < 0 then
                    call sreset(port.ssem, int(port.coun - 1));
              end;
              pid = getfirst(slist(port.rsem));
              do while not (isbadpid);
                 call kill(pid);
                 pid = getfirst(slist(port.rsem));
              end;
              call pdelete(portid);
           end;
   end pcontrol;

end ports;
$restore
```

```
/* DATA.DEC: OBJETOS, CLASES Y MONITORES.
*/

$save nolist

declare
   nclass            literally '4',    /* NUMERO MAXIMO DE CLASES          */
   nobjects          literally '16',   /*           "          OBJETOS     */
   nconditions       literally '32',   /*           "          CONDICIONES */

   isbadobjectid     literally '((oid < 0) or (oid >= nobjects))',
   isbadconditionid  literally '((cid < 0) or (cid >= nconditions))',

   clastab (nclass)  word,         /* TABLA DE IDENTIFICADORES DE CLASE    */

   objectentry       literally
   'structure (
      used    byte,              /* OBJETO ASIGNADO A ALGUNA CLÁSE?       */
      idcl    word,              /* CLASE A LA QUE PERTENECE EL OBJETO    */
      area    word,              /* SEGMENTO DE DATOS DEL OBJETO          */
      size    word,              /* TAMANO DEL SEGMENTO DE DATOS          */
      mutx    integer,           /* SEMAFOROS USADOS PARA EL ACCESO...    */
      next    integer)',         /* ...EXCLUSIVO DE LOS DATOS DEL OBJETO  */
   objctab (nobjects) objectentry,/* TABLA DE OBJETOS                     */

   conditionentry literally
   'structure (
      used byte,                 /* CONDICION ASIGNADA A ALGUN OBJETO?    */
      obid word,                 /* OBJETO AL QUE PERTENECE LA CONDICION  */
      cond integer)',            /* SEMAFORO USADO EN EL MONITOR          */
   condtab (nconditions) conditionentry;/* TABLA DE CONDICIONES          */

$restore

/* DATA.P86 (MONITOR): ADMINISTRADOR DE OBJETOS.

   OPERACIONES: OBJECTID, CONDITIONID, CWAIT, CSIGNAL, CEMPTY.
   CALIFICADORES: DOASSIGN, DOEXCLUSION, ENDEXCLUSION.
*/

$large optimize (3) save nolist

monitor: do;
$include (common.lit)
$include (es.ext)
$include (mem.ext)
$include (sem.ext)
$include (sysinit.ext)
$include (data.dec)
```

```
/* INITCLASS: INICIA LAS TABLAS DE CLASES, OBJETOS Y CONDICIONES.
*/
   initclass: procedure public;
      declare x byte;

      do x = 0 to nclass - 1;
         clastab(x) = 0;
      end;
      do x = 0 to nobjects - 1;
         objctab(x).used = false;
         objctab(x).area = 0;
      end;
      do x = 0 to nconditions - 1;
         condtab(x).used = false;
      end;
   end initclass;


/* CLASSID: INSTALA UN NUEVO IDENTIFICADOR DE CLASE EN CLASSTAB.
*/
   classid: procedure (idcl) byte;
      declare idcl word;
      declare x word, z word;

      x, z = 0;
      do while (x < nclass) and (clastab(x) <> idcl);
         if (z = 0) and (clastab(x) = 0) then z = x;
         x = x + 1;
      end;
      if x = nclass then
         if z > 0 then clastab(z) = idcl; else call sysfail (90H);
      return x < nclass;
   end classid;


/* OBJECTID: REGRESA UN NUEVO IDENTIFICADOR DE OBJETO.
*/
   objectid: procedure word public;
      declare x word;

      x = 0;
      do while (x < nobjects) and objctab(x).used;
         x = x + 1;
      end;
      if x < nobjects then objctab(x).used = true; else call sysfail (91H);
      return x;
   end objectid;
```

```
/* CONDITIONID: REGRESA UN NUEVO IDENTIFICADOR DE UNA VARIABLE CONDICION.
*/
   conditionid: procedure word public:
      declare stackposition pointer, stack based stackposition (*) word;
      declare x word;

      disable;
      stackposition = buildptr (sel (stackbase), stackptr);
      x = 0;
      do while (x < nconditions) and condtab(x).used;
         x = x + 1;
      end;
      if x < nconditions
         then
            do;
               condtab(x).used = true;
               condtab(x).obid = stack (8);
               condtab(x).cond = screate (0);
            end;
         else
            call sysfail (94H);
      enable;
      return x;
   end conditionid;


/* DOASSIGN (CALIFICADOR): ASIGNA UN OBJETO A UNA CLASE.
                           CREA UN AREA DE DATOS PROPIA PARA EL OBJETO.
                           CREA E INICIA SEMAFOROS PARA LA REGION CRITICA.
*/
   doassign: procedure (datasize) public;
      declare datasize word;
      declare stackposition pointer, stack based stackposition (*) word;
      declare oid word;

      disable;
      stackposition = buildptr (sel (stackbase), stackptr);
      oid = stack (9);
      if isbadobjectid or not objctab(oid).used
         then
            call sysfail (92H);
         else
            do;
               if classid (objctab(oid).idcl := stack (3)) then
                  do;
                     objctab(oid).size = datasize;
                     objctab(oid).area = seg (selectorof (getmem (datasize)));
                     stack (1) = objctab(oid).area;
                  end;
               objctab(oid).mutx = screate (1);
```

```
                objctab(oid).next = screate (0);
             end;
      enable;
   end doassign;


/* DOEXCLUSION (CALIFICADOR): RECUPERA EL AREA DE DATOS DEL OBJETO.
                              SOLO UN PROCESO ENTRA A LA REGION CRITICA.
*/
   doexclusion: procedure public;
      declare stackposition pointer, stack based stackposition (*) word;
      declare oid word;

      disable;
      stackposition = buildptr (sel (stackbase), stackptr);
      oid = stack (8);
      if isbadobjectid or not objctab(oid).used or objctab(oid).idcl <> stack (3)
         then call sysfail (93H);
         else
      if objctab(oid).area <> 0
         then stack (1) = objctab(oid).area;
      call wait (objctab(oid).mutx);
      enable;
   end doexclusion;


/* ENDEXCLUSION (CALIFICADOR): SALIDA DE LA REGION CRITICA.
*/
   endexclusion: procedure public;
      declare stackposition pointer, stack based stackposition (*) word;
      declare oid word;

      disable;
      stackposition = buildptr (sel (stackbase), stackptr);
      oid = stack (8);
      if isbadobjectid or not objctab(oid).used or objctab(oid).idcl <> stack(3)
         then call sysfail (93H);
      if scount (objctab(oid).next) < 0
         then call signal (objctab(oid).next);
         else call signal (objctab(oid).mutx);
      enable;
   end endexclusion;


/* CWAIT: CAUSA QUE EL PROCESO EN EJECUCION SE BLOQUEE EN UNA CONDICION.
*/
   cwait: procedure (cid) public;
      declare cid word;
      declare oid word;
```

```
      disable;
      if isbadconditionid or not condtab(cid).used
         then call sysfail (95H);
         else
            do;
               oid = condtab(cid).obid;
               if scount (objctab(oid).next) < 0
                  then call signal (objctab(oid).next);
                  else call signal (objctab(oid).mutx);
               call wait (condtab(cid).cond);
            end;
      enable;
   end cwait;


/* CSIGNAL: ACTIVA UN SOLO PROCESO BLOQUEADO EN UNA CONDICION, SI LO HAY
*/
   csignal: procedure (cid) public;
      declare cid word;
      declare oid word;

      disable;
      if isbadconditionid or not condtab(cid).used
         then call sysfail (96H);
         else
            do;
               oid = condtab(cid).obid;
               if scount (condtab(cid).cond) < 0
                  then
                     do;
                        call signal (condtab(cid).cond);
                        call wait (objctab(oid).next);
                     end;
            end;
      enable;
   end csignal;


/* CEMPTY: INDICA SI HAY PROCESOS BLOQUEADOS EN LA CONDICION.
*/
   cempty: procedure (cid) byte public;
      declare cid word;

      if isbadconditionid or not condtab(cid).used
         then call sysfail (97H);
         else return scount (condtab(cid).cond) >= 0;
   end cempty;

end monitor;
$restore
```

```
/* SYSINIT.DEC: INICIACION DEL NUCLEO DE CONCURRENCIA.
*/

$save nolist

declare
   (oldclkcs, oldclkip, oldctxcs, oldctxip) word,
/* ANTIGUO VECTOR DE INTERRUPCION DEL RELOJ (1CH)...                      */
   vecclkcs word at (0072H),      /* ...PARTE SEGMENTO                    */
   vecclkip word at (0070H),      /* ...PARTE DESPLAZAMIENTO              */
/* ANTIGUO VECTOR USADO PARA CTXSW (0FFH)...                              */
   vecctxcs word at (03FEH),      /* ...PARTE SEGMENTO                    */
   vecctxip word at (03FCH);      /* ...PARTE DESPLAZAMIENTO              */

$restore

/* SYSINIT.P86 (SYSINIT): INICIACION DEL NUCLEO DE CONCURRENCIA.

   OPERACIONES: SYSINIT, SYSEND, SYSFAIL.
*/

$large optimize(3) save nolist

sysinit: do;
$include (common.lit)
$include (es.ext)
$include (conf.dec)
$include (q.ext)
$include (mem.ext)
$include (proc.ext)
$include (sem.ext)
$include (clock.ext)
$include (data.ext)
$include (ports.ext)
$include (sysinit.dec)


/* SYSINIT: INICIACION DE LOS MODULOS DEL NUCLEO: ENTRADA/SALIDA, MEMORIA,
            LISTAS DE ESPERA, PROCESOS, SEMAFOROS, PUERTOS Y CLASES, RELOJ.
*/
   sysinit: procedure public;

      call iosysinit;
      call setcursoron;
      call clrscr;
      output(20H) = 13H;
      output(21H) = 08H;
      output(21H) = 09H;
      disable;
```

```
      oldctxcs = vecctxcs;
      oldctxip = vecctxip;
      oldclkcs = vecclkcs;
      oldclkip = vecclkip;
      enable;
      call meminit;
      call qinit;
      call procinit;
      call seminit;
      call clkinit;
      call initclass;
      call pinit;
      scrrow = 24; scrcol = 0; chratr = 71H; scratr = 7;
      call writestr(@(' PL/M-86 C:',0), cr);
      chratr = 7;
   end sysinit;


/* SYSEND: INICIO DEL AMBIENTE DE CONCURRENCIA (RESCHED).
           PROCESO NULO EN EJECUCION MIENTRAS EXISTA UN PROCESO DORMIDO,
           ELEGIBLE O EN EJECUCION. RESTAURA EL AMBIENTE PREVIO.
*/
   sysend: procedure public;

      disable;
      running = true;
      currpid = nullproc;
      clkruns = true;
      enable;
      reschedule;
      do while slnempty;
         enable;
         output(20H) = 20H;
      end;
      clkruns = false;
      disable;
      vecclkcs = oldclkcs;
      vecclkip = oldclkip;
      vecctxcs = oldctxcs;
      vecctxip = oldctxip;
      enable;
      scrrow = 24; scrcol = 0; chratr = 71H; scratr = 7;
      call writestr(@(' :PL/M-86 C',0), cr);
      scrrow = 24; scrcol = 78; chratr = 7; scratr = 7;
      call writechr(' ');
      call setcursoroff;
      call iosysterminate;
   end sysend;
```

```
/* SYSFAIL: USADO EN CONDICIONES DE ERROR.
            PRODUCE UN MENSAJE E INTENTA ABORTAR EL AMBIENTE DE CONCURRENCIA
*/
   sysfail: procedure(err) public;
      declare err word;

      scrrow = 24; scrcol = 0; chratr = 71H; scratr = 7;
      call writestr(@(' Error:',0), ' ');
      call writehex(err, '.');
      call writestr(@(' Proceso:',0), ' ');
      call writehex(unsign(getpid), ' ');
      call writestr(@('destruido',0), cr);
      output(20H) = 20H;
      enable;
      if running then
         do;
            call insert(nullproc, rdyhead, maxint - 1);
            reschedule;
         end;
   end sysfail;

end sysinit;
$restore
```

```
/* ES.P86 (ES): OPERACIONES DE LECTURA DEL TECLADO Y ESCRITURA EN PANTALLA

   OPERACIONES: SETCURSORON, SETCURSOROFF, CLRSCR, WRITECHR, WRITEBIN,
                WRITEHEX, WRITESTR, READHEX, READSTR.

   FUNCIONES DE CONVERSION: SEG, SEL.

   GENERADOR DE NUMEROS SEUDOALEATORIOS: RANO.
*/

$large optimize(3) save nolist

es: do;
$include (common.lit)

declare
   (scrrow, scrcol, scratr, chratr) byte public,
   video (2000) word at (080000H),
   (videoofs, oldofs, cursor) word;


/* SETCURSORON: HACE VISIBLE UN CURSOR PROPIO.
*/
   setcursoron: procedure public;
      cursor = 0F8DBH;
   end setcursoron;


/* SETCURSOROFF: HACE INVISIBLE EL CURSOR.
*/
   setcursoroff: procedure public;
      cursor = 0;
      chratr, scratr = 7;
      video(videoofs) = cursor;
   end setcursoroff;


/* WRITECHR: ESCRIBE UN CARACTER SOBRE EL AREA DE VIDEO MONOCROMO.
*/
   writechr: procedure(c) public;
      declare (c, d) byte;

      if (d := c) = CR
         then c = 20H;
         else
      if c = BACKSPACE
         then c = 20H;
      oldofs = videoofs;
      videoofs = double(scrrow) * 80 + scrcol;
      video(videoofs) = shl(double(chratr), 8) + c;
```

```
      if oldofs - videoofs > 1 or videoofs - oldofs > 1 then
         video(oldofs) = shl(double(scratr), 8) + 20H;
      if d = CR
         then scrcol = 79;
         else
      if d = BACKSPACE
         then scrcol = scrcol - 2;
      if (scrcol := scrcol + 1) > 79 then
         do;
            scrcol = 0;
            scrrow = scrrow + 1;
         end;
      if scrrow > 24 then
         do;
            scrcol = 0;
            scrrow = 24;
            call movw(@video(80), @video, 1920);
            call setw(shl(double(scratr), 8) + 20H, @video(1920), 80);
         end;
      videoofs = double(scrrow) * 80 + scrcol;
      video(videoofs) = cursor;
   end writechr;


/* CLRSCR: LIMPIA LA PANTALLA. PONE EL CURSOS EN LA ESQUINA SUPERIOR IZQUIERDA
*/
   clrscr: procedure public;
      scrrow = 0;
      scrcol = 0;
      scratr = 7;
      chratr = 7;
      call setw(shl(double(scratr), 8) + 20H, @video, 2000);
      video(videoofs := 0) = cursor;
   end clrscr;


/* WRITEBIN: ESCRIBE UNA PALABRA EN BINARIO.
*/
   writebin: procedure(v, c) public;
      declare v word, c byte;
      declare i byte;

      do i = 0 to 15;
         if v < 0
            then call writechr('1');
            else call writechr('0');
         v = shl(v, 1);
      end;
      call writechr(c);
   end writebin;
```

```
/* WRITEHEX: ESCRIBE UNA PALABRA EN HEXADECIMAL.
*/
   writehex: procedure(v, c) public;
      declare v word, c byte;
      declare u word, i byte;

      do i = 0 to 3;
         u = v and 0F000H;
         u = shr(u, 0CH);
         if 0 <= u and u <= 9
            then call writechr(u + '0');
            else call writechr(u - 10 + 'A');
         v = shl(v, 4);
      end;
      call writechr(c);
   end writehex;


/* WRITESTR: ESCRIBE UNA CADENA DE CARACTERES TERMINADA CON CERO.
*/
   writestr: procedure(p, c) public;
      declare p pointer, c byte;
      declare s based p (*) byte, a byte;

      a = -1;
      do while s(a := a + 1) <> 0;
         call writechr(s(a));
      end;
      call writechr(c);
   end writestr;


/* GETCHAR: LEE UN CARACTER DEL TECLADO. PROVIENE DEL MODULO IO.
*/
   getchar:procedure byte external;
   end getchar;

/* READHEX: LEE UNA PALABRA EN HEXADECIMAL.
*/
   readhex: procedure word public;
      declare v word, (a, b) byte;

      v = 0;
      do while (a := getchar) <> CR;
         call writechr(a);
         if '0' <= a  and a <= '9'
            then a = a - '0';
            else
```

```
         if 'a' <= a and a <= 'f'
            then a = a - 'a' + 10;
            else
         if 'A' <= a and a <= 'F'
            then a = a - 'A' + 10;
            else a = 16;
         if 0 <= a and a <= 15 then
            v = v * 16 + a;
      end;
      return v;
   end readhex;


/* READSTR: LEE UNA CADENA DE CARACTERES DEL TECLADO TERMINADA POR CR
*/
   readstr: procedure(p, c) public;
      declare p pointer, c byte;
      declare s based p (*) byte, (a, b) byte;

      a = -1;
      do while (b := getchar) <> CR and (b <> LF);
         call writechr(b);
         if b = BACKSPACE
            then
               if a <> -1
                  then a = a - 1;
            else s(a := a + 1) = b;
      end;
      s(a := a + 1) = 0;
      call writechr(c);
   end readstr;


/* SEG: CONVIERTE EL TIPO SELECTOR A WORD.
*/
   seg: procedure(s) word public;
      declare s selector;
      declare w word at (@s);
      return w;
   end seg;


/* SEL: CONVIERTE EL TIPO WORD A SELECTOR.
*/
   sel: procedure(w) selector public;
      declare w word;
      declare s selector at (@w);
      return s;
   end sel;
```

```
/* RAND: GENERA UN NUMERO SEUDOALEATORIO.
*/
   rand: procedure(n) word public;
      declare n word;
      declare d word;

      d = (65 * d + 33) mod 65535;
      return d mod n;
   end rand;
end es;
```

```
/* CLASS: DECLARACIONES QUE SE DEBEN INCLUIR EN EL MODULO
          QUE DEFINE A UNA CLASE.
*/

$save nolist

declare
/* DECLARA LAS SENTECIAS QUE OCULTAN LA LLAMADA A CALIFICADORES.            */
   assign     literally                          /* DO ASSIGN;              */
   ';declare mntr_ byte; call doassign(.mntr_);',
   enda       literally 'end',                   /* ENDA;                   */
   exclusion  literally ';call doexclusion;',    /* DO EXCLUSION;           */
   endx       literally 'call endexclusion; end', /* ENDX;                  */
   entry      literally 'public';

$include (common.lit)
$include (es.ext)
$include (data.ext)
$restore

/*  XINU.EXT:  DECLARACIONES DE OPERACIONES DE XINU.  ARCHIVOS INCLUIDOS  A
               TIEMPO DE COMPILACION.
*/
$include (common.lit)
$include (es.ext)
$include (conf.dec)
$include (mem.ext)
$include (data.ext)
$include (q.ext)
$include (proc.ext)
$include (sem.ext)
$include (clock.ext)
$include (ports.ext)
$include (sysinit.ext)
.HE
```

APENDICE III

ERRORES A TIEMPO DE EJECUCION

Los errores son producidos cuando se produce una condición de error cuyo número de identificación aparece a continuación. El sistema tratará de recuperarse del error producido abortando el ambiente de concurrencia.

La lista mensajes que aparece con un número de error debe interpretarse como diferentes causas posibles del error aunque no necesariamente exclusivas.

ADMINISTRADOR DE MEMORIA

0000 GETSTACK: Se solicita una cantidad negativa o nula de memoria.

0001 GETSTACK: Memoria insuficiente.

0002 FREESTACK: Extensión del bloque devuelto es cero, o el apuntador es incorrecto.

0003 FREESTACK: Apuntador de bloque alterado: se encima con otros en el area del sistema.

ADMINSITRADOR DE PROCESOS

0010 RESUME: El identificador de proceso es inválido. El proceso no está suspendido.

0011 SUSPEND: El identificador del proceso es inválido. No está en ejecución. No es elegible para recibir el procesador.

0012 KILL: El identificador del proceso inválido.

0013 CREATE: No puede crear un nuevo identificador de proceso. El tamaño de la pila es menor de 256 bytes. La dirección de la pila es ilegal. La prioridad es menor que 1.

0014 DOPROCESS: No puede crear un nuevo identificador de proceso. No puede crear la pila de ejecución.

0015 CHPRIO: El identificador del proceso es inválido. La nueva prioridad es menor que 1.

0016 GETPRIO: El identificador del proceso es inválido.

SINCRONIZACION CON SEMAFOROS

0020 WAIT: El identificador del semáforo es inválido.

0021 SIGNAL: El identificador del semáforo es inválido.

0022
0023 SCREATE: No puede crear un nuevo semáforo. La iniciación es menor que 0.

0024 SDELETE: Identificador de semáforo inválido.

0025 SRESET: Identificador de semáforo inválido. La iniciación es menor que 0.

0026 SIGNALN: Identificador de semáforo ilegal. Cuenta negativa.

0027 SCOUNT: Identificador de semáforo ilegal.

0028 SLIST: Identificador de semáforo inválido.

COMUNICACION DIRECTA

0030 SEND: Identificador de proceso ilegal.

ADMINISTRADOR DEL RELOJ

0040 SLEEP10: Tiempo de espera inválido. No hay reloj funcionando.

0041 SLEEP: Tiempo de espera ilegal. No hay reloj funcionando.

APENDICE III. MENSAJES DE ERROR

ADMINISTRADOR DE PUERTOS. COMUNICACION INDIRECTA

0050 PCREATE: No puede crear el espacio para los mensajes.

0051 PCREATE: No puede crear un nuevo identificador de puerto. El número de mensajes es negativo o cero. El tamaño del mensaje es negativo o cero.

0052 PDELETE: Identificador de puerto inválido.

0053 PCOUNT: Identificador de puerto inválido.

0054 PSEND: Identificador de puerto inválido.

0055 PRECEIVE: Identificador de puerto inválido.

0056 PWRITE: Identificador de puerto inválido.

0057 PREAD: Identificador de puerto inválido.

0058 PCONTROL: Identificador de puerto inválido. Puerto destruido.

ADMINISTADOR DE OBJETOS

0090 DOASSIGN: No puede crear un nuevo identificador de clase.

0091 OBJECTID: No puede crear un nuevo identificador de objeto.

0092 DOASSIGN: Identificador de objeto inválido o no asignado.

0093 DOEXCLUSION: Identificador de objeto inválido. El objeto no fuè asignado a la clase.

0094 ENDEXCLUSION: Identificador de objeto inválido. El objeto no fuè asignado a la clase

0095 CONDITIONID: No puede crear un nuevo identificador de condición.

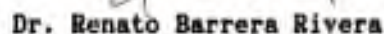
0096 CWAIT: Identificador de condición inválido.

0097 CSIGNAL: Identificador de condición ilegal.

0098 CEMPTY: Identificador de condición ilegal.

El jurado designado por la Sección de Computación del Departamento de Ingeniería Eléctrica del Centro de Investigación y de Estudios Avanzados del IPN. aprobó esta tesis el 27 de junio de 1988.

Dr. Manuel Edgardo Guzmán Rentería



Dr. Renato Barrera Rivera

M. en C. César Alejandro Galindo Legaria

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITECNICO NACIONAL

BIBLIOTECA DE INGENIERIA ELECTRICA
FECHA DE DEVOLUCION

El lector está obligado a devolver este libro antes del vencimiento de préstamo señalado por el último sello.

10 FEB. 1993
22 ABR. 1993
30 AGO. 1994
3 OCT. 1994
18 OCT. 1994
25 ABR. 1995

DEVOLUCION

AUTOR OLMEDO AGUIRRE, J.O.

TITULO PL/M-86 CONCURRENTE

CLASIF. XM 88.7 RGTRO BI 10,992

NOMBRE DEL LECTOR	FECHA PREST.	FECHA DEVOL.
Hernando Cruz Rivera	2-I-1190	[illegible]
[illegible]	[illegible]	[illegible]
Olmedo	Ago 9,94	[illegible]
Baúl Hdez Cruz	12-IX-94	[illegible]
Olmedo	4 Abr 95	[illegible]
[illegible] Dr. Guillermo [illegible] L.	09-11-95	[illegible]
Aurora - Horizonte	[illegible]	