UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

U.A.C.P. y P.

I.I.M.A.S.

CONCURRENCIA Y PARLELISMO EN ROBOTICA.

T E S I S

QUE PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A

I. GUADALUPE RODRIGUEZ GARCIA

MAYO 1998

AGRADECIMIENTOS

Al doctor Enrique Ruiz Velasco por su apoyo y guía en la elaboración de éste trabajo

A los Maestros en Ingeniería Ricardo Pastrana y Dubhe Chavira por su asesoría en al desarrollo de la parte electrónica.

INDICE

INTRODUCCIÓN

El avance en el desarrollo de sistemas operativos multiusuario, junto con el de las capacidades del hardware, obligó a los diseñadores de sistemas operativos a desarrollar técnicas para optimizar el uso de tales recursos, desde el manejo de memoria, tiempo de procesamiento, almacenamiento, impresión, etc. A fin de lograr un mejor uso de los mismos (evitando saturación o desperdicio).

Esto se logra mediante el empleo de técnicas de programación, tales como programación concurrente, en paralelo, distribuida, etc. que permiten una mejor distribución de dichos recursos,

Cuando una persona se encuentra por vez primera con conceptos como el de programación concurrente, le cuesta trabajo entenderlos, no sólo por ser un concepto nuevo para él, el hecho de que dos procesos o más estén al mismo tiempo haciendo uso de recursos, parece difícil de entender, sobre todo para los programadores que están acostumbrados a programar en forma secuencial.

En este trabajo se intenta aclarar éste tipo de conceptos, no sólo para programadores y gente que usa computadoras, sino que, se intenta introducir a personas que no usen esas herramientas a éste tipo de conceptos y tecnologías a través de la enseñanza-aprendizaje de los conceptos de programación concurrente y paralela, haciendo uso de unos dispositivos robóticos que ejemplifiquen tales conceptos.

La complejidad de la programación concurrente ha cambiado sustancialmente en los últimos años. Primero, los avances teóricos han marcado la definición de nuevas notaciones de programación que expresan de forma simple la concurrencia, hacen los requerimientos de sincronización explícitos y facilitan las pruebas formales de correctés. Segundo la disponibilidad de procesadores de bajo costo ha hecho posible la construcción de sistemas distribuidos y multiprocesadores. Con base en esos dos puntos la programación concurrente ya no es más un terreno privilegiado para aquellos que diseñan e implementan sistemas operativos; ésta se ha vuelto importante para los programadores de todas las clases de aplicaciones, incluyendo sistemas de manejo de base de datos, cálculos científicos paralelos de gran escala y sistemas de control empotrados, entre otros.

Para realizar lo anterior se ha dividido éste trabajo en 4 capítulos .

El primer capítulo presenta las definiciones de los conceptos y la teoría que permiten contextualizar y entender los mecanismos de interacción entre los procesos. Se finaliza el mismo desarrollando de manera suscinta la teoría fundamental de la programación concurrente.

En el capítulo 2 se plantea el diseño de tales dispositivos integrado por la parte electrónica (la interfaz) y por la parte mecánica, se pretende ejemplificar al algoritmo de los cinco filósofos y en modelo cliente servidor así que tales dispositivos deberán servir para tales propósitos.

En el capítulo 3 se enlistan los componentes de software que se utilizan, desde el sistema operativo, el lenguaje de programación utilizado, los programas que implantan los algoritmos concurrentes y paralelos y finalmente el programa que sirve de comunicación entre el software y el hardware.

En el último capítulo, el 4, se presentan finalmente los dispositivos desarrollados conjuntando las fases explicadas en los capítulos anteriores.

A continuación se presentan las conclusiones a las que se llegaron, desde las dificultades que se presentaron en el desarrollo del software y el hardware hasta hasta la aplicación desarrollada que se utiliza para enseñar los conceptos que se pretendían.

En la última parte de éste trabajo se presentan una serie de apéndices en los que se presentan los esquemas para el desarrollo de los dispositivos mecánicos, la interface eléctrica y un resumen de el lenguaje de programación que se utiliza.

CAPÍTULO 1

DEFINICIONES Y CONCEPTOS (TEORÍA).

Paradigmas de los lenguajes de programación[i].

Conforme el campo de los lenguajes de programación madura, su atención gira de lenguajes individuales a paradigmas que caracterizan clases de lenguajes de programación, algunos de estos paradigmas son: el distribuido y el paralelo. Los lenguajes de programación distribuida soportan cálculos a través de múltiples procesos autónomos que se comunican por paso de mensajes antes que por variables compartidas y pueden ser implementados por redes de comunicación de procesos geográficamente separadas. La programación paralela se enfoca a la ejecución concurrente, procesos que pueden competir por recursos compartidos cuando cooperan para el cumplimiento de una meta común.

El paradigma de la programación secuencial

Las aplicaciones en el procesamiento digital de información son aquellas en las que la actividad de una computadora debe incluir la reacción propia a una gran variedad de mensajes que pueden ser enviados a ella en momentos impredecibles, situación que ocurre en control de procesos, control de tráfico, aplicaciones bancarias, automatización del flujo de información en grandes organizaciones y finalmente todos los sistemas de información en los cuales un número de computadoras están acopladas una con otra.

En el uso de una computadora el progreso del tiempo se refleja en la relación "antes después" , la cual nos dice que no podemos esperar una respuesta antes de que una pregunta sea propiamente realizada. Si este comportamiento se sigue en el tiempo, es decir, haciendo preguntas y contestándolas una después de otra , se dice que se obtiene una *secuencia*. La existencia de ésta relación de orden es característica distintiva de lo que se llama un *proceso secuencial*.

El paradigma de la programación distribuida.

Bal, Steiner y Tanenbaum[ii] definen un lenguaje de programación distribuida como aquél cuyos módulos (procesos) se comunican sólo a través del paso de mensajes y no por variables compartidas. Los lenguajes para programación distribuida deben manejar paralelismo, comunicación y fallas parciales. Paralelismo puede ser expresado a nivel de procesos (Ada), objetos (Emerald), comandos. La comunicación puede ser por paso de mensajes o datos compartidos y pueden incluir no-determinismo. El paso de mensajes puede ser punto a punto, por cita o por llamada a procedimiento remoto y pueden incluir transmisión (uno a muchos). Los datos compartidos pueden ser por estructuras de datos compartidas o variables lógicas. El no determinismo

puede ser por comandos selectos o con guardia. Las fallas pueden ser manejadas por excepciones, transacciones o mecanismos de recuperación transparentes al usuario.

Los lenguajes están clasificados como completamente distribuidos, si sus módulos no comparten un espacio de direcciones y se comunican sólo mediante el paso de mensajes y lenguajes concurrentes con un espacio (global) de direcciones compartido. Los lenguajes completamente distribuidos se clasifican en términos de su soporte de sincronía en el paso de mensajes, en el paso de mensajes asíncrono, llamadas a procedimientos remotos, primitivas de comunicación múltiple, objetos y transacciones atómicas. Los lenguajes con espacio de direcciones compartidas son clasificados en lenguajes funcionales y lenguajes con estructuras de datos distribuidas.

Metodologías para la programación paralela.

Carriero y Gelertner[iii] introducen tres clases conceptuales de programación paralela : *paralelismo resultante*, el cual organiza la concurrencia por medio de computación paralela de los elementos del resultado. *El paralelismo de agenda*, el cual organiza el paralelismo de acuerdo a una agenda (gráfica pert). Y *el paralelismo de especialistas* en los cuales agentes especialistas resuelven los problemas cooperativamente. Ellos introducen tres paradigmas para implementar programas paralelos : estructuras de datos vivas las cuales se transforman así mismas en las estructuras de datos resultantes, estructuras de datos distribuidas las cuales son accesibles a muchos (pero no todos) agentes simultáneamente y el paso de mensajes el cual encapsula los datos en procesos comunicantes. El paralelismo resultante puede ser naturalmente modelado por estructuras de datos vivas, el paralelismo de agenda por estructuras de datos distribuidas y el paralelismo de especialistas por medio de paso de mensajes.

Procesos[iv]

Un *programa secuencial* especifica una ejecución secuencial de una lista de comandos; su ejecución es llamada un *proceso*. Un *programa concurrente* especifica que dos o más procesos secuenciales puede se ejecutados de manera concurrente, como procesos paralelos. Por ejemplo en un sistema de reservaciones de una línea aérea, en donde se procesan transacciones desde muchas terminales, tiene una especificación natural como un programa concurrente en la cual cada terminal es controlada por su propio proceso secuencial.

Aun cuando los procesos no son ejecutados simultáneamente, es a menudo más fácil estructurar un sistema como una colección de procesos cooperativos secuenciales antes que un programa secuencial sencillo. Por ejemplo un sistema operativo por lotes puede ser visto como tres procesos: un proceso *lector* un proceso *ejecutor* y un proceso de *impresión*. El proceso *lector* lee los datos y pone las imágenes de cada uno de éstos en un buffer de entrada. El proceso *ejecutor* lee las imágenes del buffer de entrada, ejecuta los cálculos especificados (genera imágenes de líneas) y almacena los resultados en un buffer de salida. El proceso de *impresión* obtiene las imágenes de las líneas del buffer de salida y las escribe en la impresora.

Un programa concurrente puede ser ejecutado ya sea al permitir que los procesos compartan uno o más procesadores o

ejecutando un proceso en cada procesador. La primera aproximación es conocida como *multiprogramación* y es soportada por el kernel de un sistema operativo[v] que multiplexa los procesos en el (los) procesador(es). La segunda aproximación es referida como *multiprocesamiento* si los procesadores comparten una memoria común (como en un multiprocesador[vi]o como *procesamiento distribuido* si los procesadores están conectados por una red de comunicaciones[1].

El proceso de interacción

A fin de cooperar, los procesos ejecutando concurrentemente deben comunicarse y sincronizarse. La comunicación permite que la ejecución de un proceso influya en la ejecución de otro. El proceso de comunicación está basado en el uso de variables compartidas (variables que pueden ser referidas por más de un proceso) o por el paso de mensajes.

La sincronización es a menudo necesaria cuando los procesos se sincronizan. Como los proceso son ejecutados a velocidades impredecibles. Aun para comunicarse un proceso debe ejecutar una acción que los otros detecten, tal como poner un valor en una variable o mandar un mensaje. Esto trabaja sólo si los eventos que "ejecutan una acción" o "detectan una acción", están restringidos a suceder en ese orden. Así uno puede ver la sincronización como un conjunto de restricciones en eventos ordenados. El programador emplea un *mecanismo de sincronización* para atrasar la ejecución de un proceso a fin de satisfacer tales restricciones.

[1] Un programa concurrente que es ejecutado de esta manera se le llama a menudo programa distribuido

Como ejemplo ilustrativo, consideremos el problema del sistema operativo en lotes descrito anteriormente, un buffer compartido es usado entre el proceso *lector* y el proceso ejecutor. Esos procesos deben estar sincronizados así que, por ejemplo, el proceso *ejecutor* no intentará leer la imagen de una tarjeta si el buffer está vacío.

Este tipo de sincronización viene de tomar una *aproximación operacional* de la semántica del programa. La ejecución de un programa concurrente puede ser vista como una secuencia de *acciones atómicas*, cada una resultante de la ejecución de una operación indivisible. Esta secuencia comprenderá algunos intervalos de secuencias de acciones atómicas generadas por los procesos componentes individuales. Raramente todos los intervalos de ejecución resultan en un comportamiento del programa aceptable, como se ilustra en el siguiente ejemplo, supóngase inicialmente que: $x = 0$, que el proceso *P1* incremente x en 1 y que el proceso *P2* incrementa x en 2:

P1: $x = x + 1$ *P2*: $= x + 2$

Sería razonable esperar que el valor final de x después de ejecutar concurrentemente *P1* y *P2* sería 3. Desafortunadamente, este no será siempre el caso, por que los comandos de asignación no son generalmente implementados como operaciones indivisibles. Por ejemplo la operación anterior de asignación puede ser implementada como una secuencia de tres operaciones indivisibles: i) cargar un registro con el valor de x; ii) sumarle 1 ó 2 a ella y iii) almacenar el resultado en x. Así, en el programa anterior, el valor final de x podría ser 1,2 ó 3. Este comportamiento anómalo puede ser evitado al prevenir un intervalo de ejecución de los dos comandos de asignación, esto es,

controlando el orden de los eventos correspondientes a las acciones atómicas. En otras palabras, la ejecución de *P1* y *P2* deben ser sincronizadas para forzar restricciones en los posibles intervalos.

La *aproximación axiomatica*[vii],[viii] provee un marco en el cual es posible ver el rol de la sincronización. En esta aproximación la semántica del comando está definido por axiomas y reglas de inferencia. Esto resulta en un sistema de lógica formal llamado "programación lógica." Los teoremas tiene la forma:

{P}s{Q}

y especifican una relación entre los comandos (S) y dos predicados, una *precondición* P y una *poscondición* Q. Los axiomas y reglas de inferencia se escogen de tal manera que los teoremas tienen la interpretación de que si la ejecución de S inicia en cualquier estado que satisface la precondición y su ejecución termina, entonces la poscondición será cierta. Esto permite poder ver a los comandos como una relación entre predicados.

La aproximación operacional, viendo la sincronización como un ordenamiento de eventos, es adecuada para explicar cómo trabajan los mecanismos de sincronización.

Desafortunadamente la aproximación operacional no ayuda realmente a comprender el comportamiento de un programa concurrente, aunque fue fruto de programas concurrentes simples, tiene una limitada utilidad cuando es aplicada a programas concurrentes más complejos[ix]. Esta limitante existe debido a que el número de intervalos que deben ser considerados crece exponencialmente con el tamaño de los procesos secuenciales componentes. La aproximación axiomática usualmente no tiene esta dificultad, aunque se requiere alguna familiaridad con lógica formal.

Para concretar, hay tres aspectos principales que soportan el diseño de una notación para expresar computación concurrente:

1. cómo indicar la ejecución concurrente.
2. cuál es el modo de comunicación entre procesos a usar.
3. cuál es el mecanismo de comunicación a usar.

ESPECIFICANDO EJECUCIÓN CONCURRENTE.

Corutinas:

El control es transferido entre corutinas por medio de el comando **resume**. La ejecución de un comando resume es como la ejecución de un procedimiento call, transfiere el control a la rutina nombrada, guardando la información del ambiente para controlar el regreso a la instrucción siguiente al **resume**. El control es regresado a la rutina original ejecutando otro comando resume, cualquier otra corutina puede potencialmente transferir de regreso el control a la corutina original. (Por ejemplo, la corutina C1 puede hacer un resume a la corutina C2, la cual puede hacer lo mismo a la corutina C3 y ésta, a su vez, a C1). Así el comando **resume** sirve como la única forma de transferir el control entre corutinas.

Cada corutina puede ser vista como la implementación de un proceso. La ejecución del comando **resume** genera el proceso de sincronización, cuando se usan con cuidado las corutinas, son una forma aceptable de organizar programas concurrentes que comparten un sólo procesador. Las corutinas no son adecuadas para procesamiento paralelo, debido a que su semántica permite la ejecución de solo una rutina a la vez. En esencia las corutinas son procesos concurrentes en el cual la conmutación ha sido completamente especificada.

El comando cobegin / coend

El comando **cobegin/coend** es una forma estructurada de denotar la ejecución concurrente de un conjunto de comandos. La ejecución de:

cobegin *S1*||*S2*||...||*Sn* **coend**

ocasiona ejecución concurrente de *S1*, *S2*, ..., *Sn*. Cada uno de los *Si* puede ser cualquier comando. La ejecución del cobegin termina sólo cuando la ejecución de todos los comando *Si* han terminado.

Cobegin/coend es adecuado para especificar ejecución concurrente, además, la sintaxis de cobegin hace explícito cuáles rutinas son ejecutadas de manera concurrente y provee una estructura de control de entrada y salida sencillas.

Comandos fork y join

El comando **fork**[x] especifica que una rutina designada debe empezarse a ejecutar, sin embargo, la rutina invocante y la invocada proceden concurrentemente. Para sincronizarse con el proceso invocado el proceso invocante puede ejecutar un comando **join**. La ejecución de **join** retrasa al proceso invocante hasta que la rutina invocada ha terminado:

```
proceso P1;          Proceso P2;
........             ......
fork P2              ......
......               ......
join P2;             end
...
```

La ejecución de P2 es iniciada cuando el comando fork en P1 es ejecutado; P1 y P2 entonces se ejecutan concurrentemente hasta que ya sea que P1 ejecute el comando join o P2 termine, P1 ejecuta los comandos que siguen al join.

Debido a que **fork** y **join** pueden aparecer en loops condicionales es necesario tener un entendimiento detallado de la ejecución del programa a fin de entender cuáles rutinas serán ejecutadas concurrentemente, sin embargo, cuando son usados en una forma disciplinada, los comandos son prácticos y poderosos. Por el ejemplo sistema operativo UNIX hace extenso uso de variantes de fork y join.

PRIMITIVAS DE SINCRONIZACIÓN BASADAS EN VARIABLES COMPARTIDAS

Cuando se usan variables compartidas para comunicar procesos, son útiles dos tipos de sincronización: la exclusión mútua y la sincronización condicionada. *La exclusión mútua* asegura que una secuencia de instrucciones será tratada como una operación indivisible. Considérese por ejemplo una estructura de datos compleja manipulada por medio de operaciones implementadas como una secuencia de

instrucciones. Si los procesos desempeñan operaciones de forma concurrente en los mismos datos compartidos , entonces pueden ocurrir resultados no deseados. Una secuencia de instrucciones que debe ser ejecutada como una operación indivisible es llamada una *sección crítica*. El término "exclusión mútua" se refiere a la ejecución mutuamente exclusiva de secciones críticas. Nótese que los efectos de los intervalos de ejecución son visibles sólo si dos cálculos accesan variables compartidas. Si esto es así, un cálculo puede ver resultados intermedios producidos por una ejecución incompleta de la otra. Si dos rutinas no tienen variables en común, entonces su ejecución no necesita ser mutuamente exclusiva.

Otra situación en la cual es necesario coordinar la ejecución de procesos concurrentes ocurre cuando un objeto de datos compartidos está en un estado inapropiado para ejecutar una operación en particular. Cualquier proceso que intente tal operación deberá ser retardado hasta que el estado del objeto de datos (es decir los valores de las variables que integran el objeto) cambie como resultado de otro proceso ejecutando operaciones. Este tipo de sincronización se llama condición sincronizada.

Busy-Waiting

Una forma de implementar la sincronización es tener un conjunto de procesos y probar las variables compartidas. Esta aproximación trabaja razonablemente bien para implementar condiciones de sincronización, pero no para implementar exclusión mútua. Para señalar una condición, un proceso pone el valor de una variable compartida; en la espera de tal condición, un proceso prueba repetidamente la variable hasta que encuentra que tiene el valor deseado. Debido a que un proceso espera por una condición debe probar repetidamente la variable compartida, ésta técnica para detener un proceso es la llamada *busy-waiting* y se dice que el proceso está *oscilando*.

Para implementar la exclusión mútua usando busy-waiting, los comandos que señalan y esperan por las condiciones están combinados en protocolos cuidadosamente construidos. Peterson (1981)[4] , propone una solución al problema de exclusión mútua, la cual incluye un *protocolo de entrada*, el cual ejecuta un proceso antes de entrar a su sección crítica, y un protocolo de salida, el cual se ejecuta después de ejecutar la sección crítica:

```
Proceso P1;
   loop
      Protocolo de entrada;
      Sección crítica;
      Protocolo de salida;
      Sección no crítica
   end
end
```

```
Proceso P2;
   loop
      Protocolo de entrada;
      Sección crítica;
      Protocolo de salida;
      Sección no crítica
   end
end
```

Se necesitan tres variables para realizar la sincronización. Una variable booleana *entrari* (i= 1 ó 2) que es verdadera cuando el proceso *Pi* está ejecutando su protocolo de entrada o su sección crítica. La variable *turno* guarda el nombre del siguiente proceso al cual se le permitirá la entrada a su sección

crítica; *turno* es usada cuando dos procesos ejecutan sus respectivos protocolos de entrada casi al mismo tiempo. La solución es:

```
program ejemplo_mutex;
var
   entrar1, entrar2 : Boolean initial (false, false);
   turno 0 : integer initial("P1");
   process P1;
     loop
        protocolo_de_entrada;
          entrar1:= true; { intenta entrar}
          turno := "P2";  {pone prioridad a otro proc.}
          while  entrar2 and turno = "P2"
           do skip; {espera si turno de otro proceso}
        sección_crítica;
        protocolo_de_salida
          entrar1:=false; {no intenta otra entrada}
        Sección_no_crítica;
     end;
   end;

   Proceso P2;
     loop
       protocolo_de_entrada;
         entrar2 := true; {intenta entrar}
         turno := "P1"; {poner prioridad a otro proc.}
         while  entrar1 and turno:="P1"
           do skip; {espera si turno de otro proceso}
       sección_crítica;
       protocolo_de_salida;
         entrar2:= false; {no intenta otra entrada}
       sección_no_crítica;
     end;
   end;
end
```

Además de implementar la exclusión mútua, esta solución tiene otras propiedades deseables. Primero, no ocasiona *deadlock*. Deadlock es un estado en el cual dos o más procesos están esperando por eventos que nunca ocurrirán, en el ejemplo el deadlock podría ocurrir si cada proceso pudiera girar por siempre en su protocolo de entrada, usando *turno* se previene el deadlock. La segunda propiedad deseable es la de *equidad*: si un proceso está tratando de entrar a su sección crítica él será eventualmente capaz de hacerlo, debido a que otro proceso sale de su sección crítica. En general un mecanismo de sincronización es *equitativo* si ningún proceso es detenido por siempre, esperando por una condición que ocurre de manera frecuente. El protocolo del ejemplo es equitativo, debido a que un proceso esperando entrar a su sección crítica es detenido por a lo más una ejecución de una sección crítica de otro proceso, la variable *turno* se encarga de esto.

Semáforos

Un semáforo es una variable no negativa evaluada como entero sobre la cual se definen dos operaciones: **V** y **P**. Dado un semáforo s, **P**(s) espera hasta que $s > 0$ y entonces ejecuta $s:=s-1$; la prueba y decremento son ejecutados como una operación indivisible[2]. **V**(s) ejecuta $s:=s+1$ como una operación indivisible. En la mayoría de las implantaciones de semáforos se asume que exhiben equidad: ningún proceso detenido mientras se ejecuta **P**(s) permanecerá detenido por siempre si las operaciones **V**(s) son ejecutadas de manera frecuente. La necesidad de equidad surge cuando un número de procesos son simultáneamente detenidos, todos tratando de ejecutar una operación **P** en el mismo semáforo. Claramente, la implantación debe escoger a cuál se le permitirá proceder cuando una operación **V** sea ejecutada. Una manera simple de asegurar la equidad es la de activar los procesos en el orden en el cual fueron detenidos.

---

[2] P es la primera letra de la palabra holandesa "passeren"la cual significa "a pasar"; V el la primera letra de "vrygeven"la cual significa "a liberar"

Los semáforos son una herramienta muy general para resolver el problema de sincronización. Para implantar una solución al problema de exclusión mútua, cada sección crítica es precedida por una operación **P** y seguida por una operación **V** en el mismo semáforo. Todas las secciones críticas mutuamente excluyentes usan el mismo semáforo, el cual es inicializado a uno. Debido a que tal semáforo sólo toma los valores cero o uno, es llamado semáforo *binario*.

Para implementar la condición de sincronización, las variables compartidas se usan para representar la condición y el semáforo asociado con la condición es usado para completar la sincronización. Después de que un proceso ha hecho la condición cierta, éste lo indica ejecutando una operación **V**; un proceso espera hasta que una condición sea cierta al ejecutar una operación **P**. Un semáforo que puede tomar cualquier valor no negativo es llamado semáforo *general*. Los semáforos generales son a menudo usados para sincronización condicional cuando se trata de controlar la asignación de recursos. Así un semáforo tiene como su valor inicial el número de unidades de los recursos; una operación **P** es usada para detener un proceso hasta que una unidad de recursos libres esté disponible; una operación **V** es entonces ejecutada cuando una unidad de recursos es regresada. Los semáforos binarios son suficientes para algunos tipos de sincronización condicional, especialmente para aquellos en los cuales el recurso tiene solo una unidad. El problema de la exclusión mútua se puede representar en términos de semáforos de la siguiente manera:

```
program  ejemplo_mutex;
var
   mutex : semaphore initial (1);
  process  P1;
     loop
       P(mutex);{protocolo de entrada)
         Sección_crítica;
         V(mutex);  (protocolo de salida)
         Sección_no_crítica;
         end;
       end;
    process P2;
       loop
         P(mutex); (protocolo de entrada)
         Sección_crítica;
         V(mutex); (protocolo de salida )
         Sección_no_crítica;
         end;
       end;
end.
```

Hay que notar qué tan simples y simétricos son los protocolos de entrada y salida en esta versión. En particular este uso de **P** y **V** asegura la exclusión mútua y la ausencia de deadlock. La implantación de semáforos es también equitativa y ambos procesos del ejemplo siempre salen de su sección crítica, cada proceso logra eventualmente entrar a su sección crítica.

Regiones críticas condicionales

Las *regiones críticas condicionales*[xii],[xiii] proveen una notación estructurada para especificar sincronización. Las variables compartidas son explícitamente puestas en grupos, llamadas *recursos*. Cada variable compartida debe estar en a lo más un recurso y puede ser accesada sólo en los comandos de región crítica condicional (RCC) que nombran al recurso. La exclusión mútua se da al garantizar que la ejecución de diferentes RCC, cada una nombrando al mismo recurso no se enciman. La condición de sincronización se da por las condiciones booleanas explícitas en los comandos de la RCC.

Un recurso *r* que contiene las variables *v1 ,v2,. . . , vn* es declarado como sigue:

resource *r*: *v1, v2 , . . . ,vn*

Las variables en *r* pueden ser sólo accesadas dentro de los comandos RCC que nombran a *r*. Tales comandos tienen la forma:

region *r* **when** *B* **do** *S*

donde *B* es una expresión booleana y *S* es un comando de lista. (Las variables locales al proceso en ejecución pueden aparecer también en los comandos RCC.) Un comando RCC retarda el proceso en ejecución hasta que *B* sea cierta; *S* se ejecuta entonces. La evaluación de *B* y la ejecución de *S* son ininterrumpibles por otros comandos RCC que nombran al mismo recurso. Así se garantiza que B, sea cierta, cuando la ejecución de *S* empieza. El mecanismo de retardo se asume usualmente que es equitativo: un proceso esperando una condición *B* que es repetidamente cierta se le permitirá eventualmente que continúe.

En el siguiente fragmento de código se muestra el uso de las regiones críticas condicionales, el cual es la implantación de un sistema operativo por lotes:

```
program SIS_OP;
  type buffer(T) = record;
   slots  : array [0 . .N-1] of T;
   head, tail : 0 . . N-1 initial (0,0);
   size : 0 . . N initial (0)
   end;
var
   ent_buffer : buffer(entrada)
   sal_buffer : buffer(salida)
resource b : ent_buffer; sb : sal_buffer;

process lector;
  var ent : entrada;
  loop
    lee datos de entrada;
    region eb when ent_buffer.size < N do
      ent_buffer.slots[ent_buffer.tail]:= ent;
      ent_buffer.size := ent_buffer.size + 1;
      ent_buffer.tail := (ent_buffer.tail + 1) mod N
      end
  end
end;
process ejecutor;
var ent : entrada;
    sal : salida;
loop
  region eb when ent_buffer.size > 0 do
    ent:= ent_buffer.slots[ent_buf.head]
    ent_buffer.size:= ent_buffer.size -1;
    ent_buffer.head := (ent_buffer.head + 1) mod N
    end;
   procesa entrada y genera salida
   region sb when sal_buffer. size > N do
    sal_buffer.slots[sal_buffer.tail] := sal;
    sal_buffer.size := sal_buffer  + 1;
    sal_buffer.tail := (sal_buffer.tail + 1) mod N
    end
  end
end;

process  salida;
var sal : entrada;
loop
  region sb when sal_buffer.size > 0 do
   sal:=sal_buffer.slots[sal_buffer.head];
   sal_buffer.size := sal_buffer.size - 1;
   sal_buffer.head := (sal_buffer.head + 1) mod N
  end;
  genera salida;
  end
 end
end.
```

hay que notar como las condiciones de sincronización han sido separadas de la exclusión mútua. Las expresiones booleanas en los comandos RCC que accesan los buffers explícitamente especifican las condiciones requeridas para accesar; así la exclusión mútua de diferentes comandos RCC que accesan el mismo buffer es implícita.

Cada comando RCC implanta una operación en el recurso que lo llama. Asociado con cada recurso *r* está una *relación*

invariante I_r: un predicado que indica el estado de los recursos después de que éste ha sido inicializado y después de que la ejecución de cualquier operación en los recursos y que deber ser verdadera. Por ejemplo en el segmento de código anterior las operaciones de insertar y remover los registros de los buffers *ent_buffer* y *sal_buffer* ambos satisfacen la invariante:

BI:
  0 • *head*, *tail* • *N* -1 **y**
  0 • *size* • *N* **y**
  tail = (*head* + *size*) **mod** *N* **y**
  slots[*head*]hasta *slots*[(*tail* -1) **mod** *N*]
  el buffer contiene el registro insertado más reciente en orden cronológico.

La expresión booleana *B* en cada sentencia RCC se escoge de manera que la ejecución de la lista de sentencias, cuando empezó en cualquier estado que satisface I_r y *B*, terminará en un estado que satisfaga I_r. Por lo tanto la invariante es verdadera así como no hay procesos a la mitad de la ejecución de una operación (es decir, ejecutando en una región crítica condicional asociada con el recurso). Hay que recordar que la ejecución de una región crítica condicional asociada con un objeto de datos compartido no se enciman. Aunque las pruebas de los procesos están libres de interferencia así como (1) las variables locales de un proceso aparecen sólo en la prueba de ese proceso y (2) las variables de un recurso aparecen sólo en afirmaciones dentro de las regiones críticas condicionales para ese recurso. Así una vez que las invariantes apropiadas para un recurso han sido definidas, un programa concurrente puede ser entendido en términos de sus procesos secuenciales.

Monitores

Un monitor se forma al encapsular la definición del recurso y las operaciones que lo manipulan[xiv]. Esto permite que un recurso accesado concurrentemente sea visto como un módulo[xv]. Consecuentemente, un programador puede ignorar los detalles de implantación del recurso cuando lo use y puede ignorar cómo se usa cuando se programó el monitor que lo implantó.

Un monitor consiste de una colección de *variables permanentes*, usadas para almacenar el estado de los recursos, y algunos procedimientos, los cuales implantan las operaciones en el recurso. Un monitor también tiene código de inicialización de variables permanentes, el cual se ejecuta antes que cualquier procedimiento del cuerpo sea ejecutado. Los valores de las variables permanentes son almacenados entre las activaciones de los procedimientos del monitor y pueden ser accesados sólo desde dentro del monitor. Los procedimientos del monitor pueden tener parámetros y variables locales, cada una de las cuales toma nuevos valores para cada activación del procedimiento. La estructura de un monitor se presenta enseguida:

```
monombre : monitor;
  var
  declaración variable permanentes;
  procedure op1(parámetros);
    var declaración de variables locales de op1;
    begin
      código para implantar op1;
    end;
. . .
  procedure opN(parámetros);
    var
    declaración de variable locales para opN;
    begin
      código para implantar opN
    end;
  begin
    código para inicializar variables permanentes;
  end.
```

el nombre del monitor es *mononbre* y lo componen los procedimientos *op1, . . ., opN*.

El procedimiento opJ dentro del monitor monombre es invocado al ejecutar el comando:

call *monombre.opJ*(argumentos).

La invocación tiene la semántica asociada con la llamada de un procedimiento. En suma, en la ejecución de los procedimientos en un monitor dado, está garantizado que sean mutuamente excluyentes. Esto asegura que las variables permanentes nunca serán accesadas concurrentemente.

Se usa una *variable de condición* para retrasar la ejecución en un monitor; ésta puede ser declarada sólo dentro del monitor. Dos operaciones son definidas sobre las variables de condición: **signal** y **wait** Si *cond* es una variable de condición, entonces la ejecución de:

cond.**wait**

provoca que el invocador sea bloqueado en *cond*. La ejecución de:

cond.**signal**

trabaja como sigue: si no hay un algún proceso bloqueado en *cond*, el invocador continúa; de otra manera, el invocador es temporalmente suspendido y un proceso bloqueado en *cond* es reactivado. Un proceso suspendido debido a una operación **signal** continua donde no haya otro proceso ejecutando en el monitor. Además, los señalizadores tienen prioridad sobre los procesos que están tratando de comenzar la ejecución de un procedimiento del monitor. En las variables de condición se asume equidad en el sentido de que un proceso no permanecerá suspendido por siempre en una variable de condición que es señalada frecuentemente. Hay que notar que la introducción de las variables de condición permite que más de un procedimiento esté en el mismo monitor aunque sólo uno será retardado por las operaciones wait o signal

El siguiente es un ejemplo de un monitor que define un buffer acotado:

```
type buffer(T) = monitor;
var
{las variables satisfacen la invariante IB anterior}.
  slots : array[0..N-1] of T;
  head, tail: 0..N-1;
  size: 0..N;
  notfull, notempty: condition;

procedure deposit(p: T);
  begin
    if size  = N then notfull.wait;
    slots[tail] := p;
    size := size +1;
    tail := (tail +1) mod N;
    notempty.signal;
  end;

procedure fetch(var it: T);
  begin
    if size = 0 then notempty.wait;
    it := slots[head];
    size := size -1;
    head := (head +1) mod N;
    notfull.signal;
  end;

begin
  size := 0;
  head := 0;
  tail := 0;
end.
```

A veces, un programador requiere más control sobre el orden en el cual los procesos suspendidos serán reactivados. Para implantar tal *término medio de programación*, el comando **wait** con *prioridad* puede ser usado:

cond.**wait**(*p*)

tiene la misma semántica que *cond*.**wait**, excepto que los primeros procesos bloqueados en la condición de variable *cond* serán reactivados en orden ascendente de *p*.

Un problema común que incluye el término medio de programación es el de considerar "el trabajo siguiente más corto" para la asignación de recursos. Un recurso será asignado a lo más a un usuario a la vez; si más de un usuario están esperando por el recurso cuando éste es liberado, éste es asignado al usuario que lo usará la menor cantidad de tiempo. Un monitor para implantar tal asignador se muestra enseguida. El monitor tiene dos procedimientos: (1) *pedir(tiempo: integer)*, el cual es llamado por el usuario para pedir acceso el recurso por *tiempo* unidades; y (2) *liberar*, el cual es llamado por el usuario para liberar el acceso al recurso:

```
asigna_sig_mas_corto: monitor;
var
   libre : boolean;
   turno : condition;

procedure pedir(tiempo:integer)
begin
   if not libre then turno.wait(tiempo);
   libre:= false
end;

procedure liberar;
begin
   libre:= true;
   turno.signal;
end;

begin
   libre:= true;
end.
```

PRIMITIVAS DE SINCRONIZACIÓN BASADAS EN PASO DE MENSAJES

Las regiones críticas y monitores son una evolución de los semáforos; proveen una forma estructurada de controlar el acceso a las variables compartidas. Una evolución diferente es el *paso de mensajes*, el cual puede ser visto como una extensión de los semáforos para transmitir datos así como a la implantación de la sincronización. Cuando el paso de mensajes es usado para comunicación y sincronización. Los procesos mandan y reciben mensajes en lugar de leer o escribir variables compartidas. La comunicación se realiza con base en un proceso, sobre la recepción de un mensaje, obtiene valores de algún proceso emisor. La sincronización se logra porque un mensaje puede ser recibido sólo después de que éste ha sido mandado, lo cual restringe el orden en el cual esos dos eventos ocurren.

Un mensaje se manda al ejecutar:

send *lista_de_expresiones*
   to *destinatario*.

El mensaje contiene los valores de la expresión en *lista de expresiones* al tiempo que **send** es ejecutado. El *destinatario* le da al programador control sobre dónde va a ir el mensaje y aún sobre cuál comando puede recibirlo. Un mensaje se recibe al ejecutar:

receive *lista_de_variables*
   from *emisor_fuente*

donde *lista_de_variables* es una lista de variables. El *emisor_fuente* le da control al programador de dónde viene el mensaje y también sobre cuál comando pudo mandarlo. La recepción del mensaje ocasiona, primero la asignación de los valores en el mensaje a las variables en *lista_de_variables* y, segundo, la subsecuente destrucción del mensaje.

Especificando canales de comunicación.

Tomando de manera conjunta el destinatario y el emisor se define un *canal de comunicación*. Varios esquemas han sido

propuestos para nombrar canales. El esquema de canal más simple es el se nombres de procesos para que sirvan como designadores de fuente y destino. Esto se conoce como *llamado directo*. Así:

send *card* to *ejecutor*

manda un mensaje que puede ser recibido sólo por *ejecutor*. De manera similar

receive *line* **from** *ejecutor*

permite recibir mensaje mandados sólo por ejecutor.

El llamado directo es fácil de implantar y de usar. Hace posible para un proceso el controlar los tiempos en los cuales recibe mensajes de otros procesos. El ejemplo del sistema operativo puede ser programado usando llamado directo como se muestra enseguida:

```
program SIS_OP;
process lector;
  var
   ent : entrada;
  loop
    read ent  from reader;
    send ent to executor
    end
  end;

process ejecutor;
  var
    ent : entrada;
    sal : salida;
  loop
    receive ent from reader;
    procesa ent y genera sal;
    send sal to printer
    end;
  end;

process printer;
  var
    sal : salida;
  loop
    receive sal from ejecutor;
    print sal on lineprinter
    end
  end
end.
```

El ejemplo también ilustra un paradigma importante para interacción de procesos, el pipeline. Un *pipeline* es una colección de procesos concurrentes en el cual la salida de un proceso es usado como la entrada de otro. La información fluye de manera análoga a la forma en que fluyen los líquidos en un tubo. Aquí la información fluye del proceso lector al proceso ejecutor y del ejecutor el proceso printer. El nombrado directo es adecuado para programar en pipeline.

Otro paradigma importante para el proceso de interacción es la relación *cliente/servidor*. Algunos procesos *servidores* suministran un servicio a algún proceso *cliente*. La solicitud de un servicio es realizada al mandar un mensaje a uno de los servidores. Un servidor recibe repetidamente una petición de un cliente por un servicio, realiza el servicio y (si es necesario) regresa un mensaje completo al cliente.

Desafortunadamente, el llamado directo no siempre es idóneo para la interacción cliente/servidor. Idealmente, la recepción en un servidor debería permitir recibir mensajes de cualquier cliente. Si sólo hubiera un cliente el llamado directo trabajaría bien; las dificultades aparecen cuando hay más de un cliente porque, al menos, un receptor se requeriría para cada uno. De manera análoga, si hay más de un servidor (y los servidores son idénticos), entonces el comando **send** en un cliente debería producir un mensaje que pueda ser

recibido por cualquier servidor. Nuevamente esto no puede ser realizado fácilmente con el llamado directo. Por lo tanto se requiere un esquema más sofisticado para definir canales de comunicación.

Uno de tales esquemas podría estar basado en el uso de *nombres globales*, a veces llamados *mailboxes*. Un mailbox puede aparecer como el destinatario en el comando **send** de cualquier proceso y como el origen de cualquier comando **receive** en cualquier proceso. Así el mensaje mandado a un mailbox dado puede ser recibido por cualquier proceso que ejecuta un comando **receive** nombrando a tal mailbox.

Este esquema trabaja bien para la interacción cliente/servidor. Los clientes mandan una petición de servicio a un mailbox; los servidores reciben las peticiones del mailbox. Desafortunadamente, la implantación de mailboxes puede ser un poco costosa sin una red especializada en comunicación. Cuando un mensaje se manda, éste debe ser repartido en todos los sitios donde un **receive** podría ser ejecutado en el mailbox destino; entonces, después que un mensaje ha sido recibido, todos esos sitios deben ser notificados que el mensaje no está más disponible para recibir.

Sincronización

Otra propiedad importante del paso de mensajes tiene que ver con que su ejecución puede causar una espera. Una sentencia es *no bloqueante* si su ejecución nunca detiene a su invocante; de otra forma se dice que es *bloqueante*. En algunos esquemas de paso de mensajes, los mensajes son almacenados entre el tiempo que son mandados y el tiempo en que son recibidos. Entonces si el buffer está lleno cuando se ejecuta el **send** hay dos opciones: el **send** podría esperar hasta que haya espacio en el buffer para el mensaje, o el **send** podría regresar una señal al invocante indicando que, debido a que el buffer estaba lleno, el mensaje no pudo ser enviado. Similarmente, la ejecución de un **receive**, cuando no hay mensajes disponible podría ya sea causar un retraso o terminar con una señal indicando que no había mensajes disponibles.

Si el sistema tiene un buffer no acotado, entonces nunca se detendrá un proceso cuando se ejecute un **send**. Esto es llamado *paso asíncrono de mensajes*, esto permite a un remitente estar arbitrariamente delante de un *receptor*. Consecuentemente, cuando se recibe, éste contiene información acerca del estado del emisor que no es necesariamente el estado actual. En el otro extremo, sin almacenamiento, la ejecución de un **send** siempre es detenido hasta que el correspondiente **receive** sea ejecutado; entonces el mensaje es transferido y ambos proceden. Esto es llamado *paso síncrono de mensajes*, cuando éste se usa, el intercambio de mensajes representa un punto de sincronización en la ejecución del emisor y el receptor. Por lo tanto el mensaje recibido siempre corresponderá el estado actual del emisor. Además, cuando el **send** termina, el emisor pueder tener una idea del estado actual del receptor. Entre esos dos extremos está el *paso de mensajes con buffer*, en el cual el buffer tiene acotación finita, éste permite al emisor estar adelante del receptor, pero no arbitrariamente adelante.

La forma de bloqueo de la sentencia **receive** es la más común, porque el proceso receptor a menudo no tiene mucho que hacer mientras espera a recibir el mensaje. Sin embargo, la mayoría de los lenguajes y los sistemas operativos cuentan con un **receive** no bloqueante o

un medio para probar si la ejecución de un **receive** poddría bloquear. Esto permite a un proceso recibir todos los mensajes disponibles y entonces seleccionar uno para procesar.

A veces se logra control posterior sobre cual mensaje puede ser recibido con la sentencia:

Receive lista_de_variables
From designador_origen **when** B

permite recibir sólo esos mensajes que hacen verdadera a B. Esto permite a un proceso observar el contenido de un mensaje enviado antes de recibirlo.

Un **receive** bloqueante implicitamente implementa sincronización entre el emisor y el receptor porque el receptor es detenido hasta que el mensaje es mandado. Para implementar ésta sincronización con **receive** no bloqueantes, se requiere hacer uso de busy-waiting. Sin embargo las sentencias bloqueantes de paso de mensajes pueden lograr el mismo efecto semántico que las no bloqueantes usando lo que se llama *comunicación selectiva*, los cuales se basan en comandos con guardia de Dijkstra[xvi]

En una sentencia de comunicación selectiva, un comando con guardia tiene la forma

guardia → sentencia

El guardia consiste de una expresión booleana seguida opcionalmente por una sentencia de paso de mensajes. El guardia surte efecto si al evaluarlo se tiene un resultado verdadero y la ejecución de la sentencia de paso de mensajes no ocasiona un retraso; no tiene efecto si el evaluar la expresión booleana es falsa. El guardia puede tener efecto temporalmente si la evaluación boolena es verdadera, pero la sentencia de paso de mensajes no puede ser ejecutada sin retraso.

Para ilustrar el uso de la comunicación selectiva se presenta el siguiente ejemplo, donde se implementa un proceso de almacenamiento, el cual almacena datos producidos por un proceso **productor** y que permite que esos datos sean obtenidos por un proceso **consumidor**:

```
process buffer;
var slotd: array[0..N-1] of T;
    Head, tail"0..N-1;
    Size: 0..N;
head := 0;
tail := 0;
size := 0;
do size < N; receive slots[tail] from producer →
   size := size +1;
   tail :0 (tail + 1) mod N
[] size > 0;
   send  slots[head] to  consumer →
   size :0 size -1;
   head := (head +1 ) mod  N
od
end.
```

El productor y consumidor son como sigue:

```
process  producer;
  var  stuff : T;
  loop
    generate stuff;
    send stuff to buffer;
    end;
end;

process consumer;
   var stuff: T;
   loop
     receive stuff from  buffer;
     use stuff;
     end
end.
```

Llamado a procedimientos remotos

Para programar interacciones cliente/servidor, el cliente y el servidor ejecutan dos sentencias de paso de mensajes: el cliente ejecuta un **send** seguida por un **receive**, y el servidor ejecuta un **receive** seguida por un **send**. Debido a que este tipo de interacción es muy común, se han propuesto sentencias de alto nivel que directamente la soporten. Estas se han llamado *llamadas a procedimientos remotos,* por que ellas representan una interfase: un cliente "llama" a un procedimiento que se ejecuta en una máquina remota.

Cuando se usa llamadas a procedimiento remoto, un cliente interactúa con un servidor por medio de una sentencia **call**:

call servicio(argumentos; resultados)

El servicio es realmente el nombre del canal. Si se usa llamado directo, el servicio designa el nombre del proceso servidor; si se usa puerto o mailbox, el servicio puede designar el tipo de servicio que está siendo solicitado. La llamada remota se ejecuta como sigue: el valor de los argumentos se mandan al servidor apropiado y el proceso de llamado espera hasta que el servicio ha sido ejecutado y el resultado vuelto de y asignado a los argumentos de resultado. Así el **call** de este tipo podría ser convertido en un **send** seguido inmediatamente por un **receive**.

Hay dos aproximaciones básicas para especificar un servidor en una llamada a procedimientos remotos. En la primera, el procedimiento remoto es una declaración como un procedimiento en un lenguaje secuencial:

Remote procedure service
(**in** valor_parámetros;
out parámetros_resultado)
cuerpo
end

Esta declaración de procedimiento se implementa como un proceso. En éste proceso, el servidor espera recibir un mensaje conteniendo el valor de los argumentos de algún proceso que lo llamó, asignándoles el valor de los parámetros, ejecuta el cuerpo y regresa un mensaje de respuesta conteniendo el valor de los parámetros de resultado, se tiene la sincronización resultante implícita del **send** y **receive**.

En la segunda aproximación, para especificar el lado del servidor, el procedimiento remoto es una sentencia, la cual puede ser ubicada en cualquier parte en la que se pueda poner cualquier otra sentencia. Esta sentencia tiene la forma:

Accept service(**in** valor_parámetros;
out parámetros_resultado)→ cuerpo

La ejecución de ésta sentencia hace que el servidor espere hasta que llegue un mensaje proveniente de un **call** al servicio. Entonces se ejecuta el cuerpo usando los valores de los parámetros y de cualquier otra variable accesible en el alcance de la sentencia. Hasta la terminación, un mensaje de regreso, que contiene los valores de los parámetros de resultado, se regresa al proceso que hizo la llamada. El servidor continúa con su ejecución.

Cuando se usa la sentencia **accept**, el llamado a procedimiento remoto se llama *"rendezvous"* por que el cliente y el servidor trabajan en conjunto mientras dure la ejecución del cuerpo de la sentencia accept y después continuan por rutas separadas. Una ventaja de la aproximación rendezvous es que las llamadas que el cliente realiza pueden ser atendidas al tiempo de selección del servidor; la sentencia accept puede ser

intercalada o anidadas. Una segunda ventaja es que el servidor puede lograr diferentes efectos para las llamadas al mismo servicio usando más de una sentencia **accept**, cada una con un cuerpo diferente (por ejemplo, el primer accept de un servicio podría ejecutar la inicialización). La ventaja más importante es que el servidor puede brindar más de una clase de servicio. En particular accept es usada a menudo en forma combinada con comunicación selectiva para permitir a un servidor esperar por y seleccionar una de varias peticiones de servicio[xvii], se presenta un ejemplo que ilustra el uso de la sentencia **accept** en problema de buffer acotado:

```
process buffer;
  var slots: array [0..N-1] of T;
      head, tail : 0..N-1;
      Size: 0..N;
  head:=0;
  tail:=0;
  size:=0;
do size<N;accept deposit(in value: T)→
  slots[tail]:=value;
  size:=size + 1;
  tail:=(tail + 1) mod N
[] size>0; accept fetch(out value: T)→
  value:=slots[head];
  size := size + 1;
  head := (head + 1) mod N
od
end.
```

El proceso buffer implementa dos operaciones: deposit y fetch. La primera se invoca por un productor al ejecutar:

```
call deposit(stuff)
```

La segunda se invoca por un consumidor al ejecutar:

```
call fetch(stuff)
```

Deposit y fetch son manejados por el proceso buffer en manera simétrica por que la llamada a procedimientos remotos incluye siempre dos mensajes, uno en cada dirección.

ESCRIBIENDO PROGRAMAS PARALELOS[iii]

Conceptos y métodos.

Clases conceptuales:

Se puede vislumbrar el paralelismo en términos de los *resultados* de un programa, en una *agenda de actividades* del mismo, o un conglomerado de especialistas que colectivamente constituyen el programa.

Podemos empezar con la siguiente analogía: supóngase que se quiere construir una casa. La aproximación obvia podría ser usar paralelismo, empleando mucha gente en el trabajo. Pero podría haber diversas formas en las cuales se podría emplear el paralelismo.

Primero, se puede vislumbrar el paralelismo, empezando por el producto terminado, el *resultado*. El resultado puede ser dividido en muchos componentes separados: muros frontales, traseros y laterales, paredes internas, cimentación, piso, etc. Después de dividir el resultado en componentes, se puede proceder a construir todos los componentes en forma simultánea, ensamblándolos de acuerdo a como sean terminados; se asigna un trabajador a los cimientos, otro al muro frontal exterior, otro a cada muro lateral , etc. Todos los trabajadores empiezan de manera simultánea. Todos proceden en paralelo hasta que el trabajo de un componente no pueda proceder hasta que el

de otro se haya terminado. En resumen a cada trabajador se le asigna *la producción de una pieza del resultado*, y todos ellos trabajan en paralelo mientras se lo permitan las restricciones naturales impuestas por el problema. Esta es la aproximación de *resultado paralelo*.

Otra forma de vislumbrar el paralelismo es empezando con la cuadrilla de trabajadores que construirán la casa. Hay que notar que la construcción de una casa requiere una colección de habilidades separadas. Se necesitan excavadores, cimenteros, constructores, carpinteros, etc. Se puede ensamblar una cuadrilla de trabajadores en la cual cada habilidad es representada por un trabajador especialista. Todos empiezan de manera simultánea pero inicialmente la mayoría de los trabajadores tendrán que esperar. Una vez que el proyecto está en marcha, muchos trabajadores (habilidades) serán llamados de manera simultánea: el carpintero (molduras de construcción) y los constructores de cimentación trabajarán de manera conjunta, concurrentemente, los trabajadores cimbrarán el techo mientras el plomero está instalando tuberías y el electricista mete cables, etc. Aunque un sólo carpintero haga todo el trabajo de maderería, muchas otras tareas se acoplarán y trabajarán de manera simultánea con el trabajo del carpintero. Ésta aproximación es adecuada para trabajos en línea, trabajos que requieren la producción o transformación de una serie de objetos idénticos. Si se está construyendo un grupo de casas, los carpinteros pueden trabajar en una mientras los cimentadores trabajan en otra. Ésta estrategia brindará paralelismo aún cuando el trabajo esté definido en términos de un objeto sencillo. En resumen cada trabajador esta designado a *realizar una clase específica de trabajo* y todos trabajan en paralelo hasta que las restricciones naturales impuestas por el problema lo permitan. Ésta es la aproximación *paralela especialista*.

Finalmente, se puede vislumbrar al paralelismo en términos de una agenda de actividades que deben ser realizadas a fin de construir la casa. Se empieza por escribir una agenda secuencial y se sigue en orden, pero a cada etapa se asignan varios trabajadores a la actividad que se esté desarrollando. Se necesitan los cimientos, después se necesitan las paredes, se necesita el techo, etc. Se ensambla un grupo de trabajadores generales, cada uno es capaz de realizar cualquier paso en la construcción. Primero todos construyen los cimientos; entonces el mismo grupo empieza a construir las paredes; después ellos construyen el techo, etc. En resumen, a cada trabajador se le asigna la *ayuda en la consecución de actividad de la agenda*, y todos ellos trabajan en paralelo mientras lo permitan las restricciones naturales del problema. Ésta es una aproximación de *agenda paralela*.

El punto esencial de las tres aproximaciones es que las tres representan de una manera clara las formas de pensar acerca del problema:

El resultado paralelo se enfoca en la concepción del producto terminado; el paralelismo especialista se enfoca en la constitución de la cuadrilla de trabajadores y el paralelismo de agenda se enfoca en la lista de tareas que deben ser desarrolladas.

Esas tres aproximaciones se aplican también al desarrollo del software:

1. se puede planear una aplicación paralela alrededor de la estructura

de datos producida como resultado final, obteniéndose paralelismo al calcular todos los elementos del resultado simultáneamente.

2. se puede planear una aplicación acerca de una agenda particular de actividades y después asignar muchos trabajadores en cada paso

3. se puede planear un aplicación acerca de una combinación de especialistas conectados en una red lógica de alguna clase; el paralelismo resulta de todos los nodos de la red lógica (con todos los especialistas) trabajando simultáneamente.

La pregunta que podría surgir es ¿Cómo saber qué tipo de paralelismo, qué clase conceptual, usar? Cada trabajador (o equipo) tiene una especialidad y el paralelismo aparece, en primera instancia cuando varios especialistas operan de manera simultánea, en segunda, cuando varios trabajadores idénticos, en un equipo cooperan en la agenda.

En el desarrollo de software, ciertas aproximaciones tienden a ser más naturales para ciertos problemas. La elección depende del problema a ser resuelto. En algunos casos una elección es inmediata. En otros, dos o las tres aproximaciones podrían ser igualmente naturales.

En muchos casos la manera más fácil de diseñar un programa paralelo es pensar en la estructura de datos resultante - paralelismo resultante. El programador se pregunta así mismo (1) ¿el programa producirá alguna estructura de datos de elementos múltiples (o pueden ser concebido en esos términos)? Si así es, (2) ¿Se puede especificar de manera exacta cómo depende cada elemento de la estructura resultante del resto de la entrada? Si así es, es fácil escribir un programa de paralelismo resultante. En otras palabras se quiere decir "construir una estructura de datos en tal forma, intentar determinar el valor de cada uno de los elementos de la estructura de manera simultánea, donde el valor de cada elemento es determinado por ciertos cálculos. Terminar cuando todos los valores sean conocidos." Podría ser que los elementos de la estructura son completamente independientes - no hay elementos que dependan de otros. Si así es, todos los cálculos pueden empezar de manera simultánea y proceder en paralelo. Podría ser también que algunos elementos no pudieran ser calculados hasta que se conozcan otros valores. En éste caso todos los cálculos empiezan de manera simultánea pero algunos se detienen de manera inmediata hasta que los valores que están esperando estén disponibles y puedan continuar.

Considérese el siguiente ejemplo, se tiene dos vectores de *n* elementos, A y B, se necesita calcular la suma de ambos. Un programa de paralelismo resultante podría ser como sigue: construir el vector de n elementos S; para determinar el i-ésimo elemento de S hay que sumar el i-ésimo elemento de A y el i-ésimo elemento de B. Los elementos de S son completamente independientes. Ninguna suma depende de otra. Todas las sumas pueden empezar simultáneamente y acabar en paralelo.

El paralelismo resultante es un buen punto de inicio para cualquier problema cuya meta sea producir una serie de valores con organización e interdependencia predecibles. Pero no todos los problemas satisfacen éstos criterios. Considérese, por ejemplo, un programa que produce una salida cuya

forma y formato depende de la entrada: un programa para formatear texto o traducir código en paralelo, cuya salida podría ser una cadena de bytes y (además) un conjunto de tablas, de tamaño y forma no predecible.

El paralelismo de agenda incluye una transformación o serie de transformaciones que son aplicadas en paralelo a todos los elementos de un conjunto. La expresión más flexible de éste tipo de paralelismo es el paradigma de maestro-esclavo. En un programa, un proceso maestro inicializa los cálculos y crea una colección de procesos trabajadores idénticos. Cada proceso trabajador es capaz de desempeñar cualquier paso en los cálculos. Los trabajadores buscan de manera repetida una tarea que desarrollar, desarrollan la tarea seleccionada y repiten; cuando no quedan tareas por desarrollar el programa (o el paso asociado) es terminado. El programa ejecuta de la misma manera no importa cuántos trabajadores haya, así que al menos hay uno. El mismo programa puede ser ejecutado con 1, 10 o 1000 trabajadores en tres ejecuciones consecutivas. Si las tareas son distribuidas en la ejecución, la estructura es de naturaleza balanceada, mientras un trabajador está ocupado con una tarea que lo absorbe, otro puede estar ejecutando una variedad de tareas cortas.

Por ejemplo, supóngase que se tiene una base de datos con los registros de los empleados y que se necesita identificar el empleo con la relación salario-dependientes más baja. Dado un registro Q, la función r(Q) calcula ésta relación. La agenda es simple: "Aplicar la función *r* a todos los registros en la base de datos; regresar la identidad de los registros para los cuales *r* es mínima." Se puede estructurar esta aplicación como un programa maestro-esclavo de una manera natural: El maestro llena una bolsa con objetos, cada uno representa un registro de empleado. Cada procesador trabajador extrae un registro de la bolsa, calcula *r*, y manda el resultado al maestro. El maestro guarda la pista de todos los mínimos mandados y, cuando todas la tareas hayan acabado, reporta la respuesta.

El *paralelismo especialista* incluye un programa que está concebido en términos de una red lógica. Éste surge cuando un algoritmo o un sistema a ser modelado es entendido mejor como una red en la cual cada nodo ejecuta un cálculo relativamente autónomo y las comunicaciones entre nodos siguen rutas predecibles. La red puede reflejar un modelo físico o la estructura lógica de un algoritmo. Las soluciones en estilo de red son particularmente transparentes y naturales cuando hay un sistema físico a ser modelado. Considérese, por ejemplo, un simulador de circuitos modelado por un programa paralelo en el cual cada circuito elemento es realizado por procesos separados.

Supóngase, por ejemplo, una compañía de transportes a nivel nacional, donde se necesita conocer los tiempos estimados de viaje entre dos puntos, dando las condiciones estimadas de caminos, clima y tráfico. Se podría diseñar un programa de paralelismo especialista de la siguiente manera: se incluiría un mapa del país en una red lógica; cada estado es representado por un nodo en la red. un nodo es responsable de mantener actualizadas las condiciones de viaje en un tiempo de tránsito esperado a través de ese estado. Para estimar un tiempo de travesía de un estado a otro, se planea una ruta y se incluye una representación de ésta, junto con una representación del camión. Se lleva el camión a través del primer estado, el cual estima el

tiempo de recorrido a través de ese estado, después se lleva el camión al siguiente estado en su ruta, si los hay entre el estado origen y el estado destino, donde se calcula su tiempo de recorrido para ése estado. Eventualmente el camión llega al estado destino, el cual calcula el tiempo de viaje para ése estado. Hay que notar que puede haber una gran cantidad de camiones que puede estar moviéndose a través de la red lógica en cualquier momento.

Los métodos de programación

En el paso de mensajes, se crean muchos procesos concurrentes y se encierra cada estructura de datos dentro de algún proceso; los procesos se comunican mediante el intercambio de mensajes. Cada proceso sólo puede accesar su propio conjunto local de datos privados. A fin de comunicarse, los procesos deben mandar datos de un espacio local a otro, para realizar esto, el programador debe incluir explícitamente las operaciones que mandan y reciben los datos.

Otra forma de comunicar procesos tiene que ver con construir el programa de acuerdo a la estructura de datos que saldrá como resultado. Cada elemento de ésta es implícitamente un proceso separado, el cual será un dato cuando termine. Para comunicarse, éste proceso implícito no intercambia mensajes; ellos simplemente se refieren a cada uno como elementos de la misma estructura. Así, si el proceso *P* tiene el dato *Q*, éste no manda un mensaje a *Q*; él termina dando un valor y *Q* lee ése valor directamente. Éstos son conocidos como programas de "*estructuras vivas.*"

Las aproximaciones de paso de mensajes y estructuras de datos vivas son similares en el sentido de que, en cada uno, todos los datos son distribuidos entre los procesos concurrentes; no hay estructuras globales compartidas. Aunque en el paso de mensajes los procesos son creados explícitamente por el programador, ellos se comunican explícitamente y pueden mandar valores repetidos a otros procesos. En un programa de estructura de datos viva los procesos son creados de manera implícita en el curso de construir una estructura de datos; ellos se comunican de manera implícita, al referirse a los elementos de la estructura de datos y cada proceso produce un sólo dato para ser usado por el resto del programa.

Entre los extremos de permitir que todos los datos sean absorbidos en la estructura de los procesos (paso de mensajes) o que todos los procesos mezclen con la estructura de datos (estructuras de datos vivas), hay un estrategia intermedia que mantiene la distinción entre un grupo de datos y un grupo de procesos. Debido a que los datos compartidos existen, los procesos pueden comunicarse y coordinarse al dejar datos en las estructuras compartidas. Esos son programas de "*estructuras de datos distribuidas.*"

Dónde usar cada una de las técnicas

Es claro que el paralelismo resultante se expresa naturalmente en un programa de estructuras de datos vivas. Por ejemplo, regresando al programa de suma de vectores, el centro de la aplicación es una estructura de datos viva, ésta estructura viva es un vector de *n*-elementos llamado *S*; atrapado dentro de cada elemento de *S* es un proceso que calcula *A*[*i*] + *B*[*i*] para cada *i*. Cuando un proceso termina, éste desaparece,

dejando atrás sólo el valor que fue calculado.

El paralelismo especialista es adecuado para el paso de mensajes: se puede construir un programa bajo el paso de mensajes al crear un proceso para cada nodo de la red y usando los mensajes para implantar comunicación. Por ejemplo, regresando al problema de estimación de rutas, se puede implantar cada nodo de la red lógica a través de un proceso; los camiones son representados por los mensajes. Para introducir un camión dentro de la red en el estado *X*, se manda un mensaje a ese nodo un mensaje "camión nuevo"; el mensaje incluye una representación de la ruta del camión, el nodo *X*(estado) calcula el tiempo de tránsito estimado y manda otro mensaje, incluyendo la ruta y el tiempo en ruta hasta el siguiente proceso en la ruta. Hay que notar que cuando hay muchos camiones en la red, muchos mensajes pueden converger en un proceso simultáneamente. Claramente, se necesita algún método para poner en cola los mensajes hasta que un proceso pueda atenderlo. La mayoría de los sistemas de páso de mensajes tienen algún sistema de almacenamiento interno.

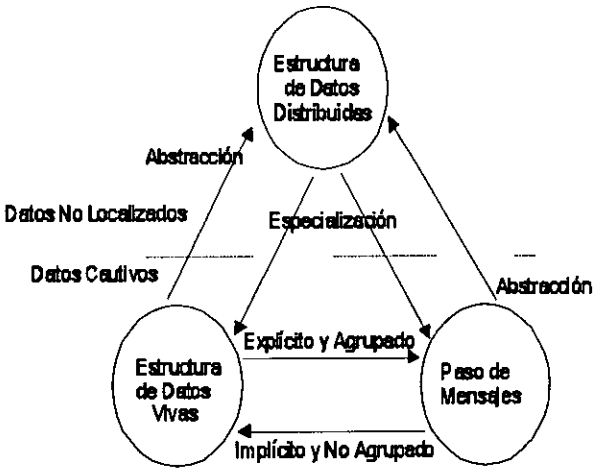
Aún cuando exista un modelo de red con esas características, el paso de mensajes será algunas veces inconveniente cuando no haya soporte de respaldo para las estructuras de datos distribuidas. Si cada nodo en la red necesita accesar a una colección de variables de estado, aquellas variables globales que pueden ser almacenadas en algunas variables locales de algún nodo, forzando que todos los accesos sean canalizados a través de un proceso custodio. Tal ordenamiento puede ser conceptualmente inepto y conducir a cuellos de botella.

El paralelismo de agenda se mapea naturalmente en métodos de estructuras de datos distribuidos. Requiere que muchos trabajadores participen en lo que es una labor sencilla. Generalmente cualquier trabajador estará esperando a que pueda conseguir una tarea. Los resultados obtenidos por algún trabajador a menudo serán necesarios para otros, pero un trabajador generalmente no sabe qué están haciendo otros trabajadores. Bajo estas circunstancias es más conveniente dejar los resultados en una estructura de datos distribuida, donde cualquier trabajador que los necesite los tome, antes que preocuparse acerca de mandar un mensaje a algún trabajador en particular.

¿Cómo se relacionan las técnicas?

Una metodología que se podría sugerir requeriría: (1) empezar con una clase conceptual que sea natural al problema, (2) escribir un programa usando el método de programación que es natural a la clase y (3), si es necesario, transformar el programa inicial en una variante más eficiente que utilice algún otro método.

Las relaciones principales se muestran en la siguiente figura.



La estructura de datos viva y el paso de mensajes se centran en *objetos de datos cautivos*: cada objeto de dato está asociado permanentemente con algún proceso. Las técnicas de estructura de datos distribuidas se centran en objetos de datos *no localizados*, es decir objetos de datos que no están asociados a algún proceso. Se puede transformar una estructura de datos viva o un programa de paso de mensajes en un programa de estructura de datos distribuida, usando *abstracción*: lo que se hace es cortar los objetos de datos de sus procesos asociados y ponerlos en una base de datos distribuida. Los procesos que no se requiere que fijen su atención en un objeto o grupo de objetos, pueden continuar libremente. Para pasar de una estructura de datos distribuida a una estructura de datos viva o programa de paso de mensajes, se usa especialización: se toma cada objeto y se liga a algún proceso.

Para pasar de la estructura de datos viva a un programa de paso de mensajes se necesita hacer comunicación *explícita* y opcionalmente usar agrupamiento. Un proceso en un programa de estructura de datos viva no necesita comunicarse explícitamente a otro proceso. El únicamente termina dejando un valor. En un programa de paso de mensajes un proceso con datos a enviar debe ejecutar una operación explícita de "envío de mensaje". Cuando un proceso de estructura de datos viva requiere un valor de un dato de algún otro proceso, él se dirige a la estructura de datos; un proceso de paso de mensajes se requerirá para ejecutar una operación de "recepción de mensajes".

La conversión de una estructura de datos viva a un programa de paso de mensajes se justifica porque la técnica de paso de mensajes ofrece cierto grado de libertad a través del agrupamiento. Un proceso en un programa de estructura de datos viva genera un valor y entonces muere. No puede seguir viviendo y generar otro valor. En el paso de mensajes, un proceso puede desarrollar tantos valores como quiera y diseminarlos en mensajes cuando quiera. Puede desarrollar una serie de mensajes durante su tiempo de ejecución. Aunque agrupado: se puede crear un proceso de paso de mensajes simple que haga el trabajo de una colección completa de procesos de estructura de datos viva.

La siguiente **tabla** resume las relaciones en diferentes formas. Presenta una caracterización general y aproximada de las tres clases. Es útil para resumir el espectro en procesos y tipos de programas.

Complejidad del proceso			
	Habilidades	Tareas	
Resultante	Uno	Uno	Simple
Especialista	Uno	Muchos	
Agenda	Muchos	Muchos	Complejo
Estructura del programa			
	Número de procesos		Coordinación
Resultante	Grande		Robusto
Especialista	Moderado		Moderado
Agenda	Ajustable		Débil

Transformación de un programa de estructuras de datos viva.

Supóngase que se tiene un problema que parece que es manejado de manera natural usando paralelismo resultante, entonces se escribe la estructura de datos viva apropiada, pero éste ejecuta de una manera pobre, así que se necesita hacer alguna transformación.

Cuando el problema parece adecuado, un programa de estructura de datos viva debe ser fácil de diseñar y expresar. Pero éste podría ejecutar de una manera pobre, debido a que la estructura de datos viva tiende a producir programas de grano fino, programas que crean una gran cantidad de procesos cada uno de los cuales hace pocos cálculos. Concretamente, si la estructura de datos resultante es una matriz de diez mil elementos esta opción creará de manera implícita diez mil procesos. Parecería que no hay razón para que este programa no trabaje de manera eficiente, pero en la mayoría de las computadoras paralelas actuales, hay una sobrecarga sustancial asociada con la creación y coordinación de un gran número de procesos. Esto es particularmente cierto en máquinas de memoria distribuída, entonces la ganancia potencial que da el paralelismo puede ser desbordada por el gran número de procesos, cada uno desarrollando un cálculo trivial.

Si un programa de estructura de datos viva no trabaja bien, un programa más eficiente es fácilmente producido a través de la abstracción hacia una versión de una estructura de datos distribuida del mismo algoritmo. Se reemplaza la estructura de datos viva con una estructura pasiva y se elevan los procesos al mismo nivel en el esquema conceptual. Cada proceso ajusta con muchos elementos, antes que con un sólo elemento. Se podrían crear cientos de procesos y cada uno de ellos calcular cientos de resultados. El programa resultante es de grano más grueso que el original, el programador decide cuántos procesos crear. Esta es una manera de prevenir la sobrecarga asociada con un gran número de procesos.

Ésta segunda versión del programa podría ser poco eficiente. Ya que requiere que cada proceso lea y escriba en una estructura de datos sencilla, cada uno de los cuales debe ser almacenado en alguna especie de memoria lógicamente compartida. El acceso a la memoria compartida será más costoso que el acceso a estructuras locales. Comúnmente éste no es un problema, pero para aplicaciones de comunicación intensiva, y particularmente en máquinas con memoria distribuida, se podría requerir de un programa más eficiente. Se podría crear una tercera versión del programa usando *especialización* para pasar de estructura de datos distribuida a paso de mensajes. Se "partiría" la estructura de datos distribuida y cada parte de ella sería procesada particularmente. En lugar de un estructura de datos distribuida se tiene ahora una colección de estructuras de datos locales, cada una encapsulada dentro de un proceso y accesible sólo por ese proceso. Cuando un proceso necesita acceso a una parte de la estructura de datos que no es la suya, debe mandar un mensaje al proceso que contiene esa parte de las estructura, preguntando por el estado de la estructura local. Esto genera un programa que no es muy agradable pero elimina las referencias directas a una estructura de datos compartida.

¿Cuándo abstraer y especializar?

¿Cómo y cuándo saber si necesitamos abstracción o pasar a un programa de paso de mensajes? La decisión es estrictamente pragmática; depende de la aplicación, el sistema de programación y de la máquina paralela.

CAPITULO 2

Diseño del Hardware

- La parte electrónica
- La parte eléctrica
- La parte mecánica

Como se mencionón en la introducción el objetivo de éste trabajo es ejemplificar conceptos como el de concurrencia y parlelismo, para lograr tal meta se desarrollan algunos dispositivos robóticos que ayuden a lograrla, para lo cual se diseñaron los filósofos y los brazos para la programación concurrente y paralela respectivamente.

Un robot puede ser definido como un sistema tecnológico, capaz de remplazar o asistir al hombre en la ejecución de una variedad de tareas físicas. El funcionamiento de un robot requiere:

1) Espacio de trabajo.

2) Una fuente de energía, constituída por las partes de trabajo y la computadora del robot.

3) Una fuente de información, programada por el hombre, la cual define el trabajo a ser ejecutado.

El trabajo del robot está asociado principalmente con la unidad central de proceso CPU y la unidad de operación.

El CPU provee la información (las señales) usando los datos que se le provean, los cuales pueden ser de tres tipos:

1) Instrucciones, definiendo la tareas a ser ejecutada (por ejemplo comandos u objetivos)

2) Medidas, concernientes al estado de la unidad operacional

3) Observaciones en el espacio de trabajo

La unidad operacional o el robot físico en sí actúan en el espacio de trabajo usando y transformando la energía adquirida desde una fuente apropiada y reaccionando a las señales provistas por el CPU. Los componentes de un robot son:

- Componentes que interactúan con el espacio de trabajo
- Partes estructurales
- Moduladores de energía
- Covertidores de energía
- Transmisores de energía mecanica
- Sensores internos[xviii]

LA PARTE ELECTRONICA[xix]

Este componente sirve de interfaz entre la pc y los dispositivos desarrollados, además se encarga de transformar los datos que se envian desde el bus hacia los dispositivos de tal manera que éstos últimos los puedan interpretar, para los cual es prudente explicar cómo es que funciona el bus de la PC.

EL BUS DE LA PC

El bus de la PC tiene 62 contactos como los bordes de la tarjeta, 31 por cada cara, este bus se divide en varios sub-buses, figura 1 apendice A.

Los sub-buses son los siguientes:

1) Alimentación: hace llegar la corriente generada en la fuente de alimentación a los componentes de la computadora.

2) Control: lleva información sobre la temporización (las señales de reloj), órdenes (memoria E/S), dirección de los datos (lectura o escritura), señales de ocupación (línea READY) e interrupciones.

3) Direcciones: lleva señales de control especiales que provocan la selección de la información a través de la computadora. Esta información se utiliza para distinguir entre los distintos dispositivos de E/S y las celdas de memoria de la computadora.

4) Datos: transporta la información a través de la computadora.

DISPOSICION Y DESCRIPCION DE LAS SEÑALES

En la misma figura1, apéndice A, se muestra la disposición de las señales en el bus de la PC. Enseguida se da una breve descripción de éstas:

CLOCK: Salida. Es el reloj del sistema. Su frecuencia depende del tipo de procesador (33, 66, 100, 133, . . .) se mide en Megahertz.

RESET: Salida. Inicializa el sistema.

D0-D7: Bus de datos. Son 8 líneas de entrada/salida.

A0-A19: Bus de direcciones. 20 líneas que determinanel máximo de memoria direccionable. A0 es el bit menos significativo y A19 el más significativo. Las señales de salida generadas por el microprocesador o por el controlador de DMA cuando éste toma el control.

IRQ2-IRQ7: Entradas. Petición de interrupción, 6 líneas que indican al procesador que algún periférico requiere su atención. IRQ2 es la señal con mayor prioridad y IRQ7 la que tiene menor.

LINEAS DE ACCESO A MEMORIA

MEMR: Salida. Indica a la memoria que el dato situado en el bus debe ser leido. Se activa en nivel bajo (0 voltios).

MEMW: Salida. Indica a la memoria que guarde el dato situado en el bus, también se activa en nivel bajo.

LINEAS DE ACCESO A DISPOSITIVOS EXTERNOS O PUERTOS

IOR: Salida. Indica a los periféricos la lectura del dato situado en el bus. Es controlada por el procesador o por el controlador DMA, se activa en nivel bajo.

IOW: Salida. Indica a los periféricos la escritura de un dato situado en el bus. Es controlada por el procesador o por el DMA, se activa en bajo nivel.

LINEAS DE ACCESO DIRECTO A MEMORIA

DRQ1-DRQ3: Entradas. Petición de DMA por los periféricos.

DACK0-DACK3: Salida. Reconocimiento de DMA, activas en nivel bajo.

AEN: Salida. Cuando se activa el DMA controla el bus de direcciones, de datos y las líneas de lectura/escritura.

T/C: Salida. Se activa al terminar el ciclo DMA.

Se dispone además de cuatro niveles de tensión de alimentación: +5, -5, +12, -12 voltios de corriente continúa.

LA TRASMISIÓN EN PARALELO

La transmisión en paralelo se utiliza en todas aquellas aplicaciones que requieren velocidad de transmisión, donde la sincronización no es un factor importante y donde los dispositivos no están muy alejados entre sí. La velocidad se puede controlar por software, la velocidad máxima en que se puede transmitir está limitada por la rapidez con que los dispositivos puedan procesar los datos.

La comunicación entre la CPU y los dispositivos se logra a través de una tarjeta de interfase que se inserta en una de las ranuras de expansión del mismo.

DESCRIPCIÓN DEL CI 8255A

El circuito integrado principal de la tarjeta que se usa en el desarrollo de la parte electrónica es el 8255A, el cual es útil para conectar la PC con dispositivos que envian bytes completos o incluso palabras de 12, 16 o 24 bits. Dicho CI tiene 24 líneas de comunicación para las cuales el programador puede definir la configuración deseada. Hay tres modos de transmisión: el modo 0 -entrada/salida básica- el modo 1 -entrada/salida habilitada- y el modo 2 -bus direccional-.

La información y los modos de operación se envian a través de la dirección de puertos de entrada/salida.

En el modo de operación 0, cada grupo de 12 terminales de E/S puede ser programado en grupos de cuatro siendo entradas o salidas. En el modo 1, cada grupo puede ser programado para tener ocho líneas de entrada y salida, de las restantes cuatro tres son usadas para intercomunicación y señales de control de interrupciones. El modo 2 es el de bus bidireccional, el cual utiliza ocho líneas como bus bidireccional y cinco para intercomunicación, traslapándose una con una del otro grupo.

FUNCIONALIDAD (los bloques del CI)

La función del 8255A es de propósito general como componente de entrada/salida en una interfase periférica del bus del CPU (figura 2 Apendice A). La configuración funcional es programada por software.

BUFFER DE DATOS DEL CI

El buffer de tres estados bidireccionales del CI se usa como interfase con el bus

de datos. El dato es transmitido o recibido por el buffer cuando la ejecución de la instrucción de entrada o salida sea realizada por el CPU. La palabra de control y la información son transferidos también por el bus de datos.

CONTROL LÓGICO Y LECTURA/ESCRITURA

Este bloque se encarga del control de las transferencias hacia adentro y afuera de datos y palabras de control, acepta entradas de las direcciones del CPU.

Las señales que se usan en este bloque son:

CS Chip Select, un nivel bajo de entrada en esta terminal habilita la comunicación entre el 8255A y el CPU.

RD ReaD un nivel bajo de entrada en esta teminal habilita el CI para enviar datos al CPU. En otras palabras le permite al CPU "leer de" el CI.

WR write, En nivel bajo habilita al CI para recibir datos, el CPU "escribe en" el CI.

A0, A1 en conjunto con las señales RD y WR controlan la selección de uno de los tres puertos o el registro de la palabra de control, normalmente están conectados a los bits menos significativos del bus de datos (A0 y A1), la siguiente tabla muestra las combinaciones:

A1	A0	RD	WR	CS	Operación de entrada (leer)
0	0	0	1	0	PUERTO A BUS DE DATOS
0	1	0	1	0	PUERTO B BUS DE DATOS
1	0	0	1	0	PUERTO C BUS DE DATOS
					Operación de salida (escribir)
0	0	1	0	0	BUS DE DATOS PUERTO A
0	1	1	0	0	BUS DE DATOS PUERTO B
1	0	1	0	0	BUS DE DATOS PUERTO C
1	1	1	0	0	BUS DE DATOS CONTROL
					INCAPACITAR FUNCION
X	X	X	X	1	BUS DE DATOS TRIESTADO
1	1	0	1	0	CONDICION ILEGAL
X	X	1	1	0	BUS DE DATOS TRIESTADO

RESET En nivel alto, esta entrada limpia el registro de control y los puertos son puestos en modo de entrada.

GRUPO DE CONTROL A Y B

La configuración funcional de cada puerto se programa vía software, por medio de una palabra de control que se envía al puerto apropiado del 8255A a través de el bus de datos (D0 a D7) y recibida por éste en su bus interno de datos.

Cada uno de éstos grupos aceptan comandos de control lógico del bus de datos interno y emiten los comandos convenientes a los puertos correspondientes.

El grupo de control A controla el puerto A y la parte alta del puerto C (C7 - C4); el grupo de control B controla el puerto B y la parte baja del puerto C (C3 - C0).

PUERTOS A, B y C

Como ya se indicó el CI tiene tres puertos de 8 bits, los cuales pueden ser configurados con características funcionales a través de software.

DESCRIPCION OPERACIONAL.

Selección de modo

Existen tres modos básicos de operación, seleccionables vía software:

MODO 0 Entrada/salida básica.
MODO 1 Entrada/salida habilitada.
MODO 2 Bus bidireccional.

Cuando la entrada RESET pasa a nivel alto todos los puertos se ponen en modo de entrada, después cuando el reset pasa a nivel bajo, los puertos permanecen en modo de entrada. Durante el tiempo de ejecución cualquiera de los otros modos pueder ser seleccionados, generando la salida apropiada.

El modo para los puertos A y B se define de manera separada, el puerto C está dividido en dos partes de acuerdo a como sea requerido por las definiciones en los puertos A y B. Los modos pueden ser combinados, por ejemplo, el puerto B puede ser programado en modo 0 mientras el puerto A puede ser programado en modo 1 (figura 3 y 4.apendice A)

Por ejemplo la palabra:

1 0 0 0 0 0 0 0

Activará todos los puertos en modo 0 (salida/salida básica)

DEFINICIÓN FUNCIONAL BASICA DEL MODO 0

- 2 puertos de 8 bits y 2 puertos de 4 bits.
- Cualquier puerto puede ser entrada o salida
- Las salidas son almacenadas
- Las entradas no son almacenadas

LA TARJETA ACOPLADORA

Para poder acceder el bus de datos de la PC a través de las ranuras de expansión, hay que utilizar determinadas direcciones de memoria. Para una tarjeta que utilice las líneas de acceso a periféricos-puertos, éstas direcciones son las que van desde 300H a 31FH. Por lo tanto se dispone de 32 direcciones para poder enviar datos por tales líneas en grupos de un byte.

Se requiere de una dirección para cada puerto más otra para la palabra de control, entonces se requiren 4 direcciones para cada tarjeta.

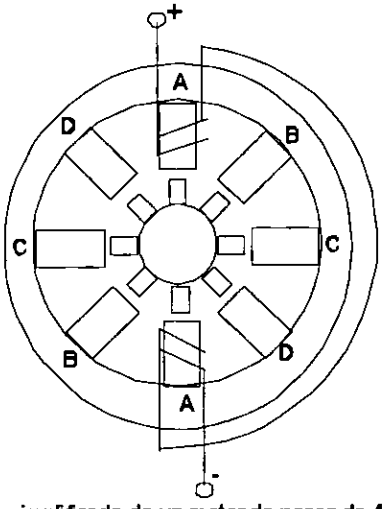
El diagrama completo del circuito electrónico acoplador puede verse en la figura 5 en el apendice A donde también se muestra la disposición de los bits de los puertos en las terminales del conector de la tarjeta.

LA PARTE ELECTRICA

La parte eléctrica, compuesta principalmente por los motores de pasos, se encarga de transformar los datos que se envían desde el bus de la PC en movimientos que activen el dispositivo robótico.

MOTORES DE PASO

Los motores de paso, son elementos especiales de la familia de los motores de corriente continua, siendo compuesto de diversos bobinados. Como se muestra en la siguiente figura:



Estructura simplificada de un motor de pasos de 4 fases

Para hacer girar el tipo de motor se aplican pulsos. Cada pulso hace que el eje se desplace un ángulo preciso (normalmente varía entre 1.8 y 7.5 grados). Con una secuencia apropiada de pulsos (ver la tabla siguiente**) se puede lograr que el eje gire en cualquier dirección y además se ubique en lugares precisos , en múltiplos del valor de paso.

Secuencia de las palabras de control para motores de 4 fases								
Secuencia de conteo	Giro a la derecha				Giro a la izquierda			
	F1	F2	F3	F4	F1	F2	F3	F4
0	0	1	0	1	0	1	0	1
1	1	0	0	1	0	1	1	0
2	1	0	1	0	1	0	1	0
3	0	1	1	0	1	0	0	1
0	0	1	0	1	0	1	0	1

En la figura anterior se muestra un motor de 4 fases, el rotor, formado por una especie de rueda dentada, se ubica siempre en relación a la bobina energizada. El intervalo entre las bobinas es el valor de paso, se obtiene pasos más cortos al incrementarse el número de bobinas.

Para calcular el valor de paso se aplica la siguiente fórmula:

X = 360 / (f . N)

Donde:
X es el valor de paso en grados
F es el número de fases
N es el número de dientes del rotor

DISEÑO MECANICO DE LOS DISPOSITIVOS

Como se ha mencionado anteriormente se ejemplificarán el agoritmo de los cinco filósofos y un modelo cliente-servidor para lo cual se diseñaron un pac-man y unos brazos mecanicos.

LA APLICACION CONCURRENTE

Las aplicaciones que se desarrollan son dos: la primera es la de un ejemplo clásico de concurrencia, el de los cinco filósofos.

Para tal aplicación se diseña un dispositivo en el cual con un motor de pasos se manipula una especie de "pac-man", el cual, a medida que el motor gira, abre y cierra la boca (Figura 1 Apéndice B).

El material con que está hecho el pac-man es lámina de acrílico la idea es que al girar el rotor, a través de una palanca haga que éste abra la boca y la cierre sucesivamente, para lo cual se implementa un "escalón" que va unido al rotor entonces, conforme gire éste, el escalón traza un círculo, pero para lograr que el pac-man abra y cierre la boca este escalón se une a la parte superior del

pac-man a través de eslabón y una palanca, el escalón pasa a través del eslabón el cual sirve para que cuando el escalón esté trazando el extremo horizontal de la circuferencia no desplace la palanca horizontalmente, esto hace que el único movimiento que se permite sobre la palanca es el vertical, para abrir y cerrar la boca. La palanca, como es de suponerse, va unida a través de un perno, a la parte superior del pac-man pero no van unidos de una forma fija, sino que la palanca tiene juego para disminuir a un más el efecto que se puede producir al estar el escalón en el extremo horizontal de la circunferencia.

Como van a ser 5 filósofos éste dispositivo se hará cinco veces, uno por cada filósofo.

Se puede tomar a cada dispositivo como un robot, por que cada uno tiene su parte electrónica, eléctrica y mecánica, o a los cinco como uno solo porque son controlados por un solo programa, el que implementa la solución al problema de los cinco filósofos.

LA APLICACIÓN PARALELA

La otra aplicación que se desarrolla es la de un brazo mecánico el cual consta de tres motores cada uno, se harán dos de éstos dispositivos.

El primer motor moverá el brazo en forma circular (primer grado de libertad), al extremo final del eje de dicho motor se encuentra el segundo motor el cual levanta o baja el antebrazo (segundo grado de libertad), el tercer motor en el extremo final del antebrazo, se encarga de subir y bajar el gancho (tercer grado de libertad). En el apéndice B figura 2 se muestra el esquema de diseño de estos dispositivos.

Como se tiene que desarrollar una aplicación paralela se arman dos brazos de éstos

EL material con que están hechos dichos dispositivos es el mismo con el que se hicieron los filósofos, el cual se eligió por ser ligero (así no se incrementa el peso de los dispositivos y consecuentemente se disminuye la potencia que tienen que desarrollar los motores) y de fácil manipulación para realizar los cortes que se requieran.

Capítulo 3

El software

El software está compuesto por el sistema operativo que se utiliza, el lenguaje de programación que se utiliza y por la implementación de los algoritmos que controlarán los movimientos de los dispositivos.

EL SISTEMA OPERATIVO LINUX[m1]

Linux es un Sistema Operativo (SO) para PCs basadas en Intel, diseñado y construido por cientos de programadores alrededor del mundo. La meta ha sido crear un clon de UNIX, de acceso gratuito que cualquier persona puede usar. En realidad Linux empezó como un hobie de **Linus Torvalds** cuando era estudiante en la universidad de Helsinky en Finlandia. Su meta era crear un SO para usarlo en lugar de el SO Minix, un SO parecido a UNIX disponible para PCs basadas en Intel.

Los beneficios derivados de usar el SO UNIX, y de ahí Linux, son producto de su poder y flexibilidad, resultado de las muchas características con las que está hecho el sistema, tales características son:

Multitarea:

La palabra *multitarea* describe la habilidad de aparentemente ejecutar múltiples programas al mismo tiempo sin dificultar la ejecución de las demás aplicaciones. Esto se ha llamado *multitarea preferente* por que se garantiza que cada programa tiene la oportinudad de ejecutarse, cada programa se ejecuta hasta que el SO le da preferencia, esto para permitir que otro programa se ejecute. Este tipo de multitarea es la que linux maneja. En otras palabras, el microprocesador puede hacer una sola cosa a la vez, pero es capaz de hacerla en periodos de tiempo muy cortos, por ejemplo un micropocesador opera con velocidades de reloj de 100Mhz, lo que esto significa es que es capaz de transferir 100 millones de bits por segundo, cuando se procesa un conjunto completo de instrucciones, las velocidades son muy elevadas. La mente humana no es capaz de diferenciar entre una corta espera y algo que ocurra simultáneamente. Parece que ejecutan al mismo tiempo.

Los beneficios de la multitarea preferente se reflejan en la disminución de tiempo muerto (el tiempo en el que no se puede continuar por que un proceso no ha terminado aun), la flexibidad de no tener que cerrar una aplicación antes de abrir y ejecutar otra es más conveniente.

Multiusuario:

Con la capacidad de LINUX para dividir el tiempo de procesamiento en muchas aplicaciones a la vez se presta para que se pueda atender a más de un usuario a la vez, cada uno ejecutando una o más aplicaciones. La característica verdaderamente sobresaliente de Linux es que más de una persona puede trabajar en la misma aplicación en la misma versión al mismo tiempo, desde terminales separadas.

Shells programables:

Aunque algunas versiones de Linux incluyen más de un tipo de shell, ellos trabajan de la misma manera. Un shell trabaja como un intérprete entre el usuario y el núcleo. La diferencia principal entre los tres shells disponibles (Bourne, C, bash) estriba en la sintáxis en línea de comando.

La programación shell sirve en tantas funciones como personas haya que la utilicen, muchas usan ésta característica para personalizar su sistema y hacerlo mas amigable para el usuario, otras la encuentran útil para eficientar muchas de las aplicaciones que ejecutan, ejecutando algunos procesos en "background" y asi poder trabajar con otros procesos

Independencia de dispositivos:

Ya que Linux es un clon de UNIX éste también tiene un kernel adaptable, conforme más programadores se unan al proyecto Linux, más dispositivos se agregan al núcleo, como último recurso, debido a que el código fuente está disponible, se puede modificar el kernel para trabajar con nuevos dispositivos.

Comunicaciones y Red:

La superioridad de UNIX sobre otros SOs es también evidente en las utilerías de comunicación y red, las cuales se heredan en Linux, ningún otro SO incluye tales capacidades de red. Y ningún otro SO ha sido desarrollado con esas características. Ya sea que se necesite comunicar con alguien através de un mail o bajar archivos grandes desde otro sistema, Linux provee los medios para realizar estas actividades.

EL LENGUAJE DE PROGRAMACIÓN SR[xxii]

El lenguaje de programación SR ha estado desarrollándose desde hace varios años. La primera versión (RS_0) contenía mecanismos para el paso asíncrono de mensajes y rendezvous. Su forma de rendezvous proveía mecanismos por medio de los cuales un proceso sirviendo un rendezvous podría escoger cuál invocación atender basado en los valores de los parámetros al momento de la invocación. La versión 1 (SR_1) proveía mecanismos adicionales para el llamado remoto a procedimientos, creación dinámica de procesos y semáforos, así como mecanismos para especificar distribución de módulos de programa.

La versión 2 (SR_2)conserva muchas de las estructuras de la versión 1. Sin embargo también maneja los mecanismo que permiten compartir objetos. Esta característica es importante en ambientes de memoria compartida, para las cuales las primeras versiones de SR no estaban planeadas (también es importante para soportar librerías, por ejemplo las de matemáticas y las de windows)

SR soporta muchas de las características para la programación concurrente, pero conservando un lenguaje sencillo y de fácil uso, proveiendo al mismo tiempo una implementación eficiente.

EL MODELO DE COMPUTACIÓN SR.

Un programa en SR puede ejecutar en múltiples espacios de dirección, los cuales pueden estar en múltiples máquinas físicas. Los procesos dentro de un solo espacio de memoria pueden compartir también objetos. Así SR soporta programación en ambientes distribuidos así como también en ambientes de memoria compartida.

El modelo de computación SR permite que un programa esté dividido en uno o más espacios de direcciones de memoria llamados *máquinas virtuales*. Cada máquina virtual define un espacio de direcciones en una máquina física. Las máquinas virtuales se crean dinámicamente, pueden ser referenciadas a través de *variables de capacidad*. Las máquinas virtuales contienen instancias de dos tipos relacionados de componentes modulares: *globales* y *recursos*.

Cada uno de esos componentes contiene dos partes: una especificación (spec) y una implementación (body). Instancias de recursos se crean dinámicamente, a través de una sentencia de creación explícita. Esas instancias y los servicios que proveen se refencían indirectamente a través de variables de capacidad. Las instancias de globales también se crean dinámicamente. Sin embargo, se crean implícitamente de acuerdo a como se vayan necesitando, específicamente, cuando no exista ya una instancia de esa global en la máquina virtual, cada máquina virtual puede contener solamente una instancia de una global. Las globales y sus servicios pueden ser referenciados a través de sus nombres.

La spec de una global o recurso puede contener declaraciones de tipos, constantes y operaciones; la spec de una global puede contener adicionalmente declaraciones de variables. Una operación define un servicio que deber ser proveido en algún lugar dentro del programa. Este puede ser considerado como una generalización de un procedimiento: tiene nombre, puede tomar parámetros y puede regresar valores. Una operación declarada en la spec de un recurso debe ser atendido en el cuerpo de ése recurso. Similarmente, una operación declarada en la spec de una global puede ser atendida en el cuerpo de la global; ésta puede ser atendida también dentro de un recurso o global que los importe.

El cuerpo de una global o recurso puede contener declaraciones de objetos adicionales; esos objetos son visibles solo dentro del cuerpo, no pueden ser importados. Los cuerpos pueden contener también código que, entre otras cosas, atiende operaciones. El código se divide en unidades llamadas *processes*[1] y *procs*. Los processes se crean implicitamente cuando la global o recurso que lo contiene se crea. Las instancias de procs se crean cuando son invocados; ellos pueden ejecutar como processes independientes. Todos los processes creados dentro de una global o recurso ejecutan en la misma máquina virtual en la cual el recurso o global que lo contiene se haya creado. Los processes y los procs pueden declarar variables adicionales y operaciones; ellos deben contener el código que atiende tales invocaciones.

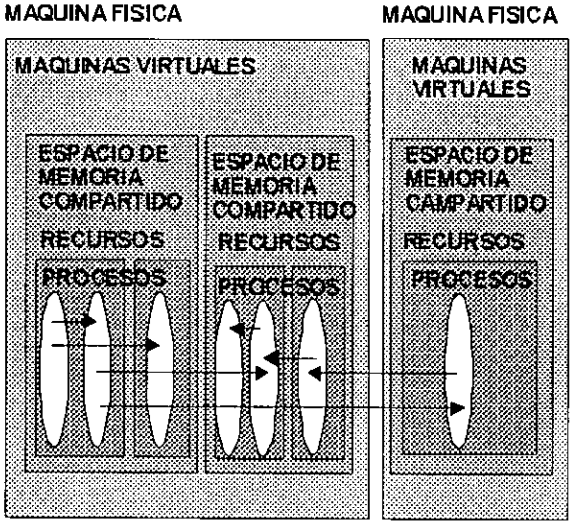
La siguiente figura resume el modelo de computación de SR. En su forma más simple, un programa consiste de una máquina virtual sencilla ejecutando en una máquina física: un programa puede consistir también de múltiples máquinas virtuales ejecutando en múltiples máquinas físicas. Formas hibridas son posibles y a veces útiles. Datos y procesadores son compartidos dentro de una máquina virtual; diferentes máquinas virtuales pueden ser puestas en diferentes máquinas físicas.

Los processes en la misma máquina virtual se pueden comunicar a través de una operación de invocación. Las operaciones

---

[1] Se deja la palabra processes para no causar confusión con las palabras proc y procedure ya que en SR cada una de ellas define una unidad lógica de ejecución distinta.

pueden ser invocadas directamente a través de el nombre declarado de la operación o a través de una variable con capacidad de recurso, o puede ser invocada indirectamente a través de una variable con capacidad de operación.



Esas variables de capacidades son robustamente tipificadas y pueden apuntar a operaciones con rúbricas estructuralmente equivalentes. Ellas pueden ser pasadas como parámetros a las operaciones durante la invocación o a los recursos durante la creación del recurso. Permitiendo tener processes en diferentes instancias de recursos o posiblemente en diferentes máquinas virtuales.

La comunicación entre processes es independiente de la localidad de la máquina virtual. Por ejemplo, el mensaje pasando entre processes en la misma instancia del recurso tiene la misma sintáxis y semántica que los mensajes pasando entre diferentes máquinas virtuales.

EL LENGUAJE

SR es rico en en variedad de mecanismos que provee para programación concurrente: creación dinámica de procesos, semáforos, paso de mensajes, llamadas a procedimientos remotos y rendezvous. Sin embargo, todas ellas se proveen a través de un sencillo mecanismo: la operación.

La idea principal es que las operaciones pueden ser invocadas de dos maneras, en forma síncrona (call) o asíncrona (send) y pueden ser atendidas en dos formas por procs o por sentencias input (in). Esto brinda las siguientes combinaciones:

Invocación	Servicio	Efecto
Call	Proc	Llamada a procedimiento (posiblemente remoto)
Call	In	Rendezvous
Send	Proc	Creación dinámica de procesos
Send	In	Paso asíncrono de mensajes

Una virtud de este enfoque es que permite la declaración de una operación en forma separada del código al que atiende. Esto permite que especificaciones de recursos y globales sean escritas y usadas sin importar cómo es que la operación es atendida.

SOPORTE PARA COMPARTIR

SR provee soporte para compartir en diversos niveles. Primero, los processes dentro de una instancia de un recurso pueden compartir variables. Ellos pueden coordinar el acceso a las variables compartidas a través de semáforos compartidos o de otras operaciones declaradas dentro del recurso. Segundo, los processes que ejecutan en posiblemente diferentes instancias de recursos pero en la misma máquina virtual pueden compartir variables y operaciones declaradas en el spec de la global.

Considérese por ejemplo, un programa escrito para ejecutar en múltiples procesadores de memoria compartida. Este puede ser escrito como un programa con un solo recurso, con processes compartiendo

variables y operaciones declaradas al nivel de recurso. Aunque para un programa de cualquier complejidad, dividir el programa en múltiples recursos es deseable. Esta clase de estructura es posible también. Los recursos pueden ser creados en una sola máquina virtual, con variables compartidas y operaciones declaradas en una o más globales.

SOPORTE PARA DISTRIBUCION

Las máquinas virtuales son las unidades básicas para programación distribuida. Pueden ser creadas (o destruidas) dinámicamente como sea necesario en la ejecución del programa. Instancias del recurso o global pueden ser creadas en máquinas virtuales. Los processes en máquinas virtuales se comunican con otros processes al invocar las operaciones.

La invocación de operaciones exibe dos clases de transparencia. Primero, una operación se invoca en la misma forma sin importar como está distribuido el programa. Las invocaciones por el cliente de las operaciones en el servidor permanecen iguales sin importar que el cliente y el servidor estén localizados en la misma máquina virtual o física. Segundo, una operación se invoca en la misma manera sin importar cómo se atiende a la misma, es decir por una sentencia input o por un proc.

En el apendice c se presenta un resumen más campleto acerca del lenguaje (incluyendo la sintaxis de los comandos y tipos de datos).

EL PROGRAMA PARA CONCURRENCIA

EL PROBLEMA DE LOS CINCO FILÓSOFOS

El problema de los cinco filósofos es propuesto por Dijkstra.[xxiii] Este problema es interesante por que en él surgen los aspectos de problemas de asignación de recursos con que los sistemas operativos distribuidos tiene que lidiar. Prevenir el deadlock y la inanición (falta de parcialidad) son las metas principales en la solución de este problema.

En el problema de los cinco filósofos, *n* filósofos sentados alrededor de una mesa redonda con *n* tenedores, uno entre cada dos filósofos. Cada filósofo come y piensa en forma alternativa. Para comer, un filósofo debe primero adquirir los tenedores que están a su derecha y a su izquierda. Después de comer el filósofo regresa los tenedores a la mesa.

Como ya se ha mencionado antes la primera aplicación que se va a desarrollar es la de los cinco filósofos para el cual se utiliza el siguiente programa:

```
resource Sirviente
  op obtent(), sueltat()
body Sirviente(id: int)
  process server
    do true ->
      receive obtent(); receive sueltat()
    od
  end
end

resource Filosofo
  import Sirviente
  external escribe_puerto(puerto : int; dato : int)
body Filosofo(l, r: cap Sirviente; id, t, n: int)
  var d, iz : int
  const no_vtas : int := 10
  process phil
    fa i := 1 to t ->
      l.obtent(); r.obtent()
      if id = 5  -> iz := 1
      [] else ->  iz := id +1
      fi
    # 48 es el número de pasos necesarios para realizar
    # una vuelta, se harán diez vueltas.
      fa j := 1 to 48*no_vtas
      # se manda la palabra al motor, interface en "C"
      escribe_puerto(puerto[id], palabra[(j mod 4) +1])
      af
      write("Filosofo",id,"tiene los tenedores",iz,id)
      write("Filosofo", id, "Est comiendo")   # come
      nap(100) #espera un poco
      l.sueltat(); r.sueltat()
      write("Filosofo", id, "Esta pensando") # piensa
    af
  end
end

resource Main()
  external inicia_tarjeta()
  import Filosofo, Sirviente
  var n, t: int
```

```
  writes("Cuantos Filosofos"); read(n)
  writes("Cuantas sesiones por filosofo")
  read(t)
  inicia_tarjeta() #incializa tarjeta y puertos
  var s[1:n]: cap Sirviente
  # crea los  Sirvientes
  fa i := 1 to n ->
    s[i] := create Sirviente(i)
  af
  # Se crean los filosofos; para prevenir deadlock,
  # Se les pasa las capacidades para sus sirvientes
    fa i := 1 to n-1 ->
    create Filosofo(s[i], s[i mod n + 1], i, t, n)
  af
  create Filosofo(s[1], s[n], n, t, n)
end
```

En esta versión de solución del problema de los cinco filósofos se utiliza un sirviente por tenedor, con lo que se evita el deadlock. Cada filósofo interactúa con dos sirvientes para obtener los tenedores que necesita, un filósofo que tenga hambre podrá comer después de obtener los tenedores que necesita.

Se utilizan tres recursos: Sirviente, Filósofo y Main, En el recurso Main se crean las *n* instancias de cada Filósofo y de cada Sirviente y pasa capacidades de ésta último a cada Filósofo para que se puedan comunicar.

Al recurso Filósofo se le pasan capacidades de dos Sirvientes e invoca *obtent* y *sueltat* en cada uno de esos Sirvientes.

El proceso *server* en cada una de las instancias del Sirviente atiende continuamente invocaciones de *obtent*, primero y después de *sueltat* de sus dos instancias de Filósofo asociados. Esto asegura que el tenedor está en a lo más un filósofo a la vez. Un filósofo se le permite comer cuando obtiene los tenedores de cada uno de los sirvientes.

Cuando el recurso Main crea instancias de Filósofo, se les pasa capacidades para sus sirvientes izquierdo y derecho. El último filósofo obtiene su tenedor del primer sirviente, así se previene el deadlock ya que cada filósofo obtiene un tenedor y después pide el otro.

EL PROGRAMA PARA PARALELISMO

MODELO CLIENTE SERVIDOR

Se utiliza un modelo cliente-servidor para implementar la aplicación paralela.

Se tienen dos brazos uno como cliente y el otro como servidor, lo que harán los dispositivos es basicamente:

El cliente solicita al servidor una pieza por lo que le envía un mensaje, el servidor al recibirlo empieza a mover la pieza, se pueden presentar las siguientes situaciones, dependiendo de la velocidad con la que trabajen los brazos:

- El cliente y el servidor trabajan al mismo tiempo, es decir con la misma velocidad.
- El servidor es más rápido que el cliente por lo que tiene que esperar a que el cliente le solicite otra pieza
- El cliente es más rápido que el servidor por lo que éste debe manejar una "bolsa de solicitudes pendientes"

El programa en SR para implementar el algorítmo anterior es el siguiente:

```
global datos
  var palabra[4][6] :
int := [[012x, 07x, 03x, 09x, 012x, 06x ],
        [120x, 70x, 30x, 90x, 120x, 60x),
        [06x, 012x, 09x, 03x, 07x, 012x),
        [60x, 120x, 90x, 30x, 70x, 120x)]
  var puerto[4] : int := (0304x, 0305x, 0306x, 0307x)
```

```
   var No_vtas1 := 10
   var No_vtas2 := 15
   var No_vtas3 := 10
   var No_pasos := 48
   var No_piezas := 2
   var veloc1 := 5
   var veloc2 := 7
body datos
   getarg(1, veloc1); getarg(2, veloc2)
end
resource brazo
   external escribe_puerto( puerto : int; palabra :
int)
   import datos
   op control(id : int; velocidad : int; sentido : int)
body brazo()
   var indice : int
   proc control(id, velocidad)
      velocidad := int(round(velocidad * 0.5) * 10)
      write("recibiendo Identidad y Velocidad:", id,
velocidad)
  # Baja antebrazo
  fa indice := 1 to No_vtas2 * No_pasos ->
    escribe_puerto(puerto[2], palabra[id,  ((indice mod
6) + 1)])
    nap(velocidad)
  af
  escribe_puerto(puerto[2], 00)

#Baja mano
  fa indice := 1 to No_vtas1 * No_pasos ->
    escribe_puerto(puerto[1], palabra[id,  ((indice mod
6) + 1)])
    nap(velocidad)
  af
  escribe_puerto(puerto[1], 00)
#Sube mano
  fa indice := 1 to No_vtas1 * No_pasos ->
    escribe_puerto(puerto[1], palabra[id + 2,  ((indice
mod 6) + 1)])
    nap(velocidad)
  af
  escribe_puerto(puerto[1], 00)
#Sube Antebrazo
  fa indice := 1 to No_vtas2 * No_pasos ->
    escribe_puerto(puerto[2], palabra[id + 2,  ((indice
mod 6) + 1)])
    nap(velocidad)
  af
  escribe_puerto(puerto[2], 00)
#Establece el sentido de giro del hombro
  giro := sentido
  if sentido = 1 then giro := id + 2
  fa indice := 1 to No_vtas3 * No_pasos ->
    escribe_puerto(puerto[3], palabra[giro,  ((indice
mod 6) + 1)])
    nap(velocidad)
  af
  escribe_puerto(puerto[3], 00)
  end
end
resource main()
   external inicia_tarjeta(control : int)
   import brazo, datos
   op solicitud(no_sol : int), respuesta(no_sol : int)
   # var maquina1, maquina2 : cap vm
   var brazo1, brazo2 : cap brazo
   var indice : int
   var  no_sol : int
   # maquina1 := create vm()
   # maquina2 := create vm()
   brazo1 := create brazo() # on maquina1
   brazo2 := create brazo() # on maquina2
   inicia_tarjeta(80x)
   process cliente
      indice := 1
      do  indice <= No_piezas ->
           send solicitud(indice)
           write(" Solicitando pieza numero:", indice)
           receive respuesta(indice)
           write(" Recibiendo solicitud numero:",
indice)
           call brazo1.control(1, veloc1,0)
           indice +:= 1
         send brazo1.control(1, veloc1, 1)
     od
   end
   process servidor
  #    var  no_sol : int
      do true ->
           receive solicitud(no_sol)
         write ("Recibiendo solicitud numero:", no_sol)
           call brazo2.control(2, veloc2, 0)
           send  respuesta(no_sol)
           write ("Se envia respuesta numero:", no_sol)
         send brazo2.control(2, veloc2, 1)
      od
   end
   inicia_tarjeta(00x)
end
```

Como se menciona anteriormente se va a implementar un modelo cliente-servidor, para lo cual se crean 2 recursos y una global, la cual contiene los datos.

El primer recurso que se ejecuta es el main, en el cual se crean las instancias los recurso para los brazos y se inicializa la tarjeta.

En en éste recurso se se tiene los procesos cliente y servidor el primero de de ellos se encarga de solicitar las piezas, con lo cual se pasa el control al servidor (que está en escucha constante) éste recibe la solicitud, mueve la pieza y regresa el control al cliente esto se hace hasta que índice sea igual al número de piezas que se van a mover. Pero como es necesario que para mover de nueva cuenta una pieza el brazo debe regresar a su posición inicial, es necesario mandar las señales ahora en sentido inverso.

Nótese que el brazo se llama primero con un call y después con un send. Esto hace que en la primera llamada la ejecución de proceso sea síncrona desde el punto de vista de quien la llama, en el segundo caso la llamada al proceso brazo se realiza a través de un send, esto hace que la ejecución del procesos sea asíncrona desde el punto de vista de quién llama al proceso, el efecto de éstas combinaciones en el llamado al proceso brazo es que mientras un brazo mueve la pieza solicitada, el otro regresa a su posición inicial

El recurso brazo se encarga de controlar el movimiento de los brazos, el primer ciclo for baja el antebrazo, el segundo baja la mano, el tercero sube la mano, el

cuarto sube el antebrazo, antes de ejecutar el quinto for se establece el sentido en que va a girar el hombro y finalmente éste gira.

En la global se declaran los datos que controlan el movimiento de los brazos tales como el número de vueltas para cada motor y la velocidad a la que se van a mover los motores

LA INTERFACE ENTRE LA APLICACIÓN Y LA TARJETA

LAS FUNCIONES EN "C"

```
#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>

#include <unistd.h>
#include <stdio.h>

int presente;


int inicia_tarjeta(void)
{
   int paso;
   if (!(paso=ioperm(0x304,4,1))) {
                     /*permiso para accesar puerto*/
     presente++;
     outb(0x80, 0x307); /* Si permiso mandar control */
    }
   else {
      printf("No se puede inicializar la tarjeta\n");
      return 1;
   }
}


escribe_puerto(unsigned short puerto,
               unsigned char dato)
{
   if (!presente)
       printf("No se han activado los puertos de la
               tarjeta\n");
   else
       outb(dato, puerto) ;
/* Escribimos dato en puerto, aquí se manda la palabra
al motor */
}
```

En la función *inicia_tarjeta* se verifica que los puertos estén accesibles, a partir de la direccion 0x304 se activan los siguiente tres siendo un total de cuatro puertos.

Si se pudieron activar los puertos se incrementa la bandera *presente* para indicar que los puertos fueron activados y que la tarjeta está presente. Después se manda la palabra de configuración de puertos al puerto control, con ésta palabra se activan los tres puertos como de salida. Si no se pueden incializar los puertos se manda el mensaje pertinente y se termina la ejecución.

Para mandar palabras a los puertos asociados a los motores se usa la función *escribe_puerto* con parámetros número de puerto y dato que se va a escribir en dicho puerto. Si *presente* es diferente de cero indica que los puertos ya tiene permiso de escritura, por lo que mandamos el dato al puerto.

Estas funciones son únicamente para accesar los puertos, el control de frecuencias de escritura a puertos lo hace el programa en SR.

Capítulo 4

Conjuntando los componentes

Hasta el capítulo anterior se han descrito y explicado cada uno de los componentes de las aplicaciones pero en forma separada, por lo que ahora se explica cómo es que trabajarán en conjunto para controlar los dispositivos robóticos con los que ejemplificarán los conceptos.

Como se mencionó en el capítulo anterior el control en la frecuencia con que se mandan los datos a los puertos de salida lo lleva a cabo el programa para la aplicación concurrente o paralela, escritos en SR, por lo que empezamos explicando el funcionamiento común para las dos aplicaciones.

En el capítulo II se dio la explicación del funcionamiento de la tarjeta que realiza el control de los puertos, cada tarjeta ocupa cuatro puertos, uno de control, para configurar el funcionamiento de los restantes, para activar e incializar el CI 8255A. En la tarjeta que se utiliza el número de puerto de control es el 0x307.

En la función *inicia_tarjeta*, presentada en el capítulo anterior se tiene la siguiente instrucción:

```
ioperm(0x304,4,1)
```

La cual activa los puertos, empezando desde al puerto 0x304 hasta el 0x307, con lo cual se activan (el tercer parámetro) los tres puertos para datos y el de control (4 en total)

Otra instrucción que también se encuentra en ésta función es:

```
outb(0x80, 0x307)
```

La cual manda la palabra 10000000 (en binario) al puerto de control, esta palabra le indica que ponga los tres puertos como salida (modo 0) y que ponga la bandera de modo en activo igual a 1 (ver figura 4 en apendice A).

Para mandar las palabras que accionan a los motores se tiene la función *escribe_puerto*, la cual recibe dos parámetros el número de puertos y el datos que se va a mandar a él:

```
escribe_puerto(unsigned short puerto,
               unsigned char dato)
```

Además se tiene la instrucción que manda el dato al puerto:

```
outb(dato, puerto)
```

Donde dato es alguna de las palabras para control de paso de los motores, un ejemplo sería:

```
outb[1](0x05, 0x304)
```

La cual manda la palabra 00000101, 0x05 en binario, (ver tabla de secuencia de palabras de control en el capítulo II) al puerto A (0x304), así para controlar dos motores con un solo puerto la parte

[1] la sintaxis y valores de retorno de outb y ioperm se explican en el manual en línea de Linux

superior de la palabra podría controlar un motor y la parte baja controlaría el otro.

Para controlar un motor de pasos de cuatro fases conectamos 4 bits del puerto de salida de la tarjeta a las cuatro bobinas del motor de paso, un bit para cada bobina, (ver figura 5 en apendice A), al enviar la secuencia de palabras mostrada en la tabla mencionada anteriormente se logra la activación del motor, cada uno en las palabras es un voltaje que se manda a las bobinas.

Pero el bit de salida no se conecta directamente a la bobina, se tiene que hacer a través de una fase de potencia que otorga la corriente necesaria para exitar las bobinas[xxiv], el diagrama de la fase de potencia y los nombres de los dispositivos necesarios se presentan en la misma figura 6 del mismo apéndice.

COMO SE ACTIVAN LOS MOTORES DESDE LOS PROGRAMAS EN SR:

En las aplicaciones el recurso Main tiene las siguientes líneas de código

```
external inicia_tarjeta()
inicia_tarjeta() #incializa tarjeta y puertos
```

la primera, justo después de la declaración del recurso, nótese la palabra reservada **external**, ésta le indica a SR que esa función no está escrita en SR y que va a ser importada. La segunda es donde se ejecuta la llamada, así es como se inicializa la tarjeta y como se hace la liga entre SR y "C".

En algunos los recursos se tienen las siguientes líneas de código:

```
external escribe_puerto(puerto : int; dato : int)
      # se manda la palabra al motor, interface en "C"
      escribe_puerto(puerto[id], palabra[(j mod 4) +1])
```

En la primera línea también se encuentra la palabra external, por que la función escribe_puerto, al igual que en el caso anterior, va a ser traida a SR. Se tiene un ciclo el cual controla la forma en que se van mandando las palabras al puerto a través de la función *escribe_puerto*, para que un motor complete una vuelta se necesita mandar doce veces la secuencia completa de palabras de control lo que da un total de 48 señales mandadas al puerto, con cada señal el motor se mueve un paso de 7.5 grados.

El puerto, en la aplicación de los cinco filósofos se selecciona de acuerdo a la identidad del filósofo que esté comiendo y como se deben de mandar en forma secuencial las cuatro palabras de control del motor con (j mod 4) + 1 se logra que se repita el ciclo de palabras.

En la aplicación paralela el puerto se selecciona de acuerdo al motor que controla qué movimiento y el brazo que lo esté realizando

CONCLUSIONES

Conclusiones

Se crearon los dipositivos necesarios para mostrar los conceptos que se deseaban exponer, en el primer caso ls ejecución concurrente y la paralela en el segundo.

En el primer caso se puede considerar a cada uno de los dispositivos como un solo robot, por lo cual tendríamos 5 robots, cada uno de ellos controlado por su propio programa (las instancias del recursos filósofo), pero como todos los "filósofos" son controlados por un solo programa (el principal), puede considerarse a los cinco como un sólo robot.

No sucede lo mismo con la otra aplicación ya que lo que se pretendía enseñar es paralelismo por lo que cada dispositivo tiene su propio programa que lo controla, cada uno dentro de su propia máquina virtual.

Cada dispositivo puede ser considerado como un robot por que cada uno tiene:

- Un componente eléctrico.
- Un componente mecánico
- Un componente electrónico
- Un componente de control

Además cada uno puede trabajar en forma individual.

Para el desarrollo del tipo de aplicaciones que se querían realizar se necesitaba un sistema operativo con capacidad de multiproceso y que además trabajará en un computadora personal con procesador Intel o compatible, entonces la elección lógica fue el sistema operativo LINUX, el cual resultó ser fácil de instalar y de usar además de ser flexible. Cada distribución viene con su código fuente el cual puede ser modificado de acuerdo a las necesidades del usuario. Este es un sistema operativo completo con capacidades similares a las de cualquier sistema operativo UNIX.

El lenguaje que se usa, SR, es un lenguaje que es fácil de entender y por lo tanto de usar, que puede ser usado para aplicaciones secuenciales, concurrentes y distribuidas.

Cuando se usa en aplicaciones secuenciales se usan los procedimientos y es muy parecido a cualquier otro lenguaje con esa orientación. Para aplicaciones concurrentes cuenta con una serie de estructuras de control que permite el desarrollo rápido y sencillo de aplicaciones, algunas de estas estructuras son: creación dinámica de procesos, semáforos, paso de mensajes, llamadas a procesos remotos, rendezvous.

Haciendo combinaciones de tales estructuras se puede tener operaciones invocadas en forma síncrona o asíncrona.

Mediante el uso de máquinas virtuales (de las cuales se puede crear varias en una máquina física) una en cada máquina física, varias en varias máquinas físicas, se pueden crear aplicaciones distribuidas. El llamado a procesos o procedimientos en una máquina virtual en otra máquina física es como si se llamara al proceso o procedimiento dentro de la

misma máquina virtual, es decir el llamado se hace de la misma forma.

La tarjeta que se desarrolló, permite la transmisión en paralelo a través de los tres puertos de lectura/escritura con los que cuenta, con una palabra de control, enviada al puerto de control se puede configurar la tarjeta para que trabaje en el modo que se desea (3 modos). La lectura/escritura de los puertos, es tan sencillo como mandar una instrucción de lectura o escritura al puerto en el cual se desea realizar la acción, así si por ejemplo se desea activar en un puerto los pines 1, 3, 5, es necesario mandar la palabra 10101000 a dicho puerto. De ésta manera la lectura/escritura se hace de manera directa sólo es necesario saber cuál pine es el que se desea activar, sabiendo lo anterior el control de motores de paso es directo y como se tiene tres puertos y un motor se controla con cuatro pines, es posible controlar seis motores con una tarjeta.

Finalmente conjuntando todos los componentes es como se desarrollaron lo componentes robóticos que, como en éste caso, trabajan concurrentemente o en paralelo, y si se dispone de los recursos necesarios se puede crear aplicaciones distribuidas.

Algunos de los métodos de programación que se utilizaron en el desarrollo de las aplicaciones fueron:

- Creación dinámica de procesos
- Llamada a procedimientos remotos
- Regiones críticas a través de las instrucciones send / receive
- Paso asincrono de mensajes con send / receive
- Un modelo cliente servidor
- Y por supuesto la ejecución concurrente.

La trasmisión de datos en paralelo hacia los puertos de la tarjeta controladora es muy rápida pero puede ser controlada por medio de programación, por lo que la velocidad de movimientos en los dispositivos debe se tomada en cuenta para que se puedan apreciar con claridad los métodos de programación antes mencionados.

Finalmente acoplando las características de todos los componentes usados se logra obtener dispositivos trabajando en forma concurrente y paralela, aplicando las mismas herramientas es posible desarrollar aplicaciones para otras áreas como el procesamiento distribuide señales, bases de datos concurrentes, ejecución calculos matemáticos en paralelo como multiplicación de matrices, sumas vectoriales, solución de sistemas de ecuaciones etc.

APENDICE A

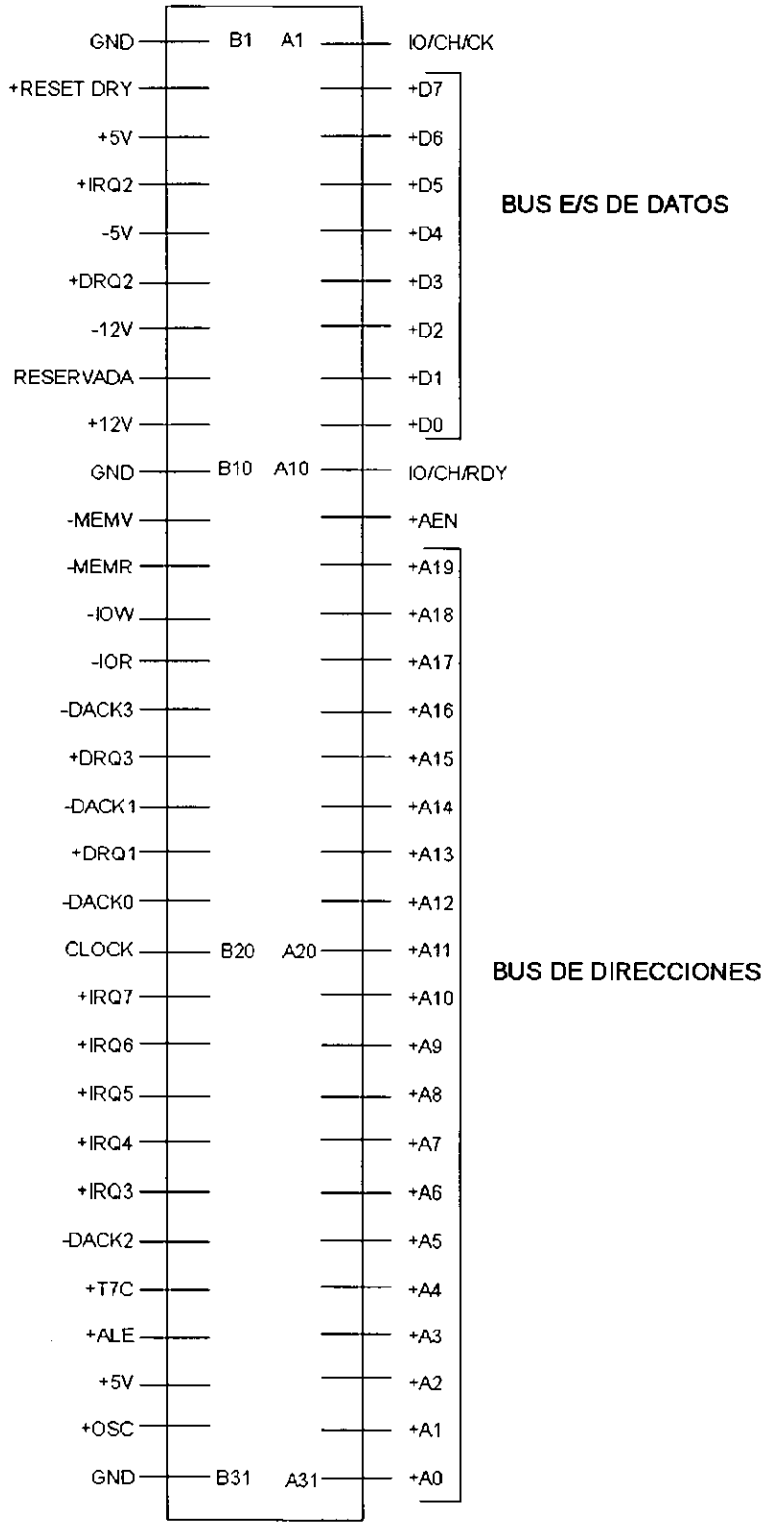


FIGURA 1: DISPOSICION DE LAS SEÑALES EN EL BUS DE LA PC
EN UNA RANURA DE EXPANSION

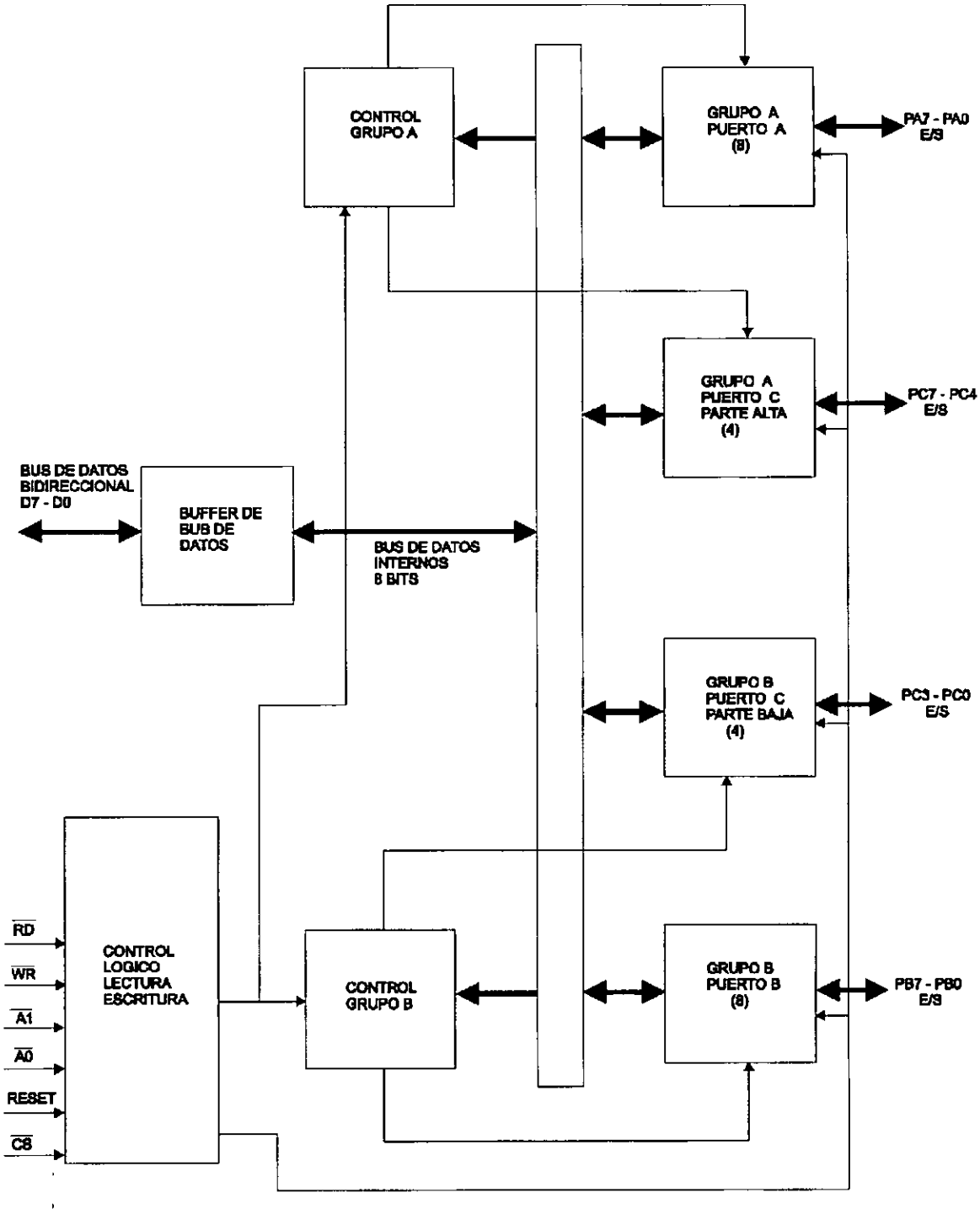


FIGURA 2 DIAGRAMA DE BLOQUES DEL CI 8255A

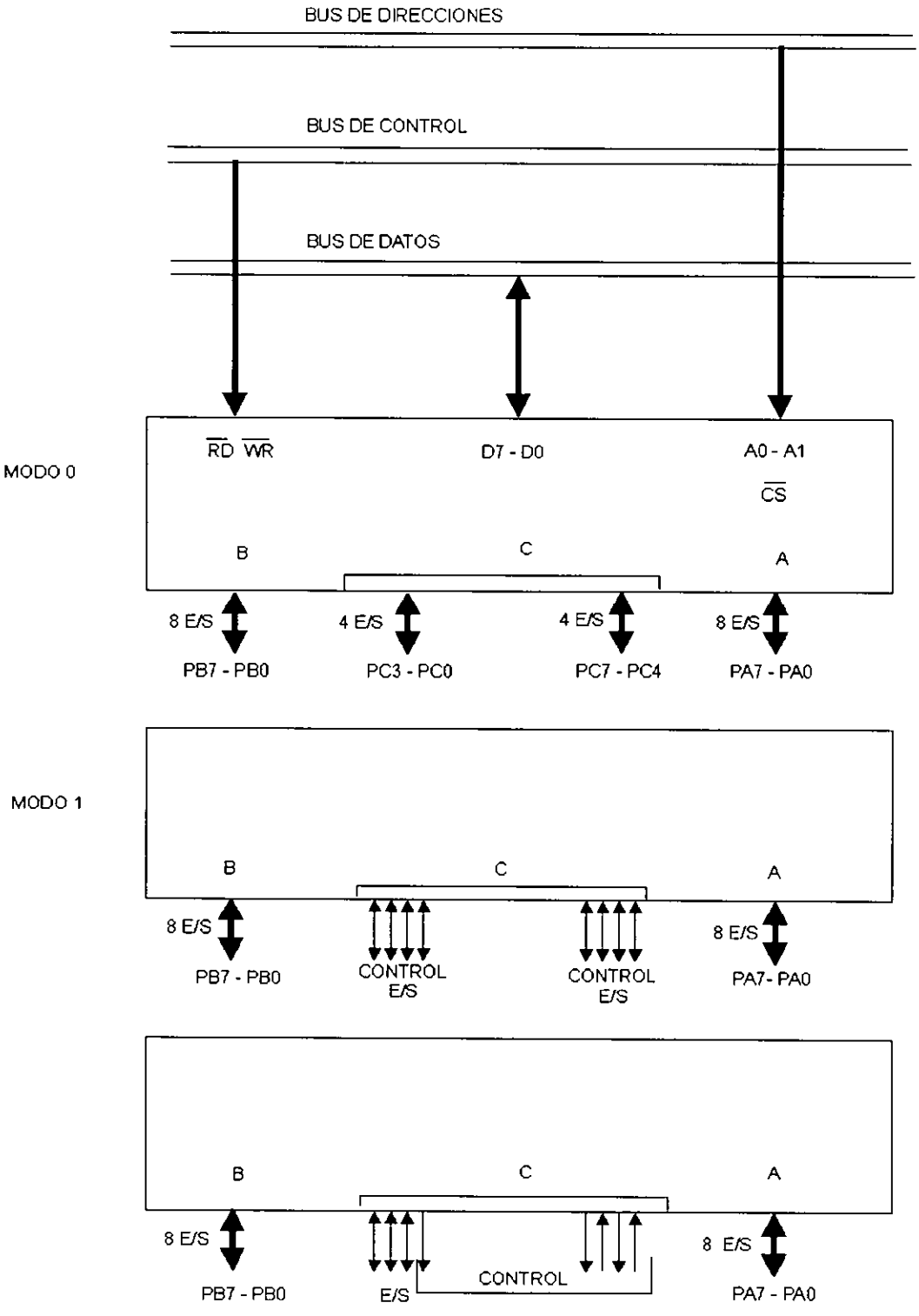


FIGURA 3 MODOS BASICOS DE OPERACION Y LA INTERFAZ CON EL BUS

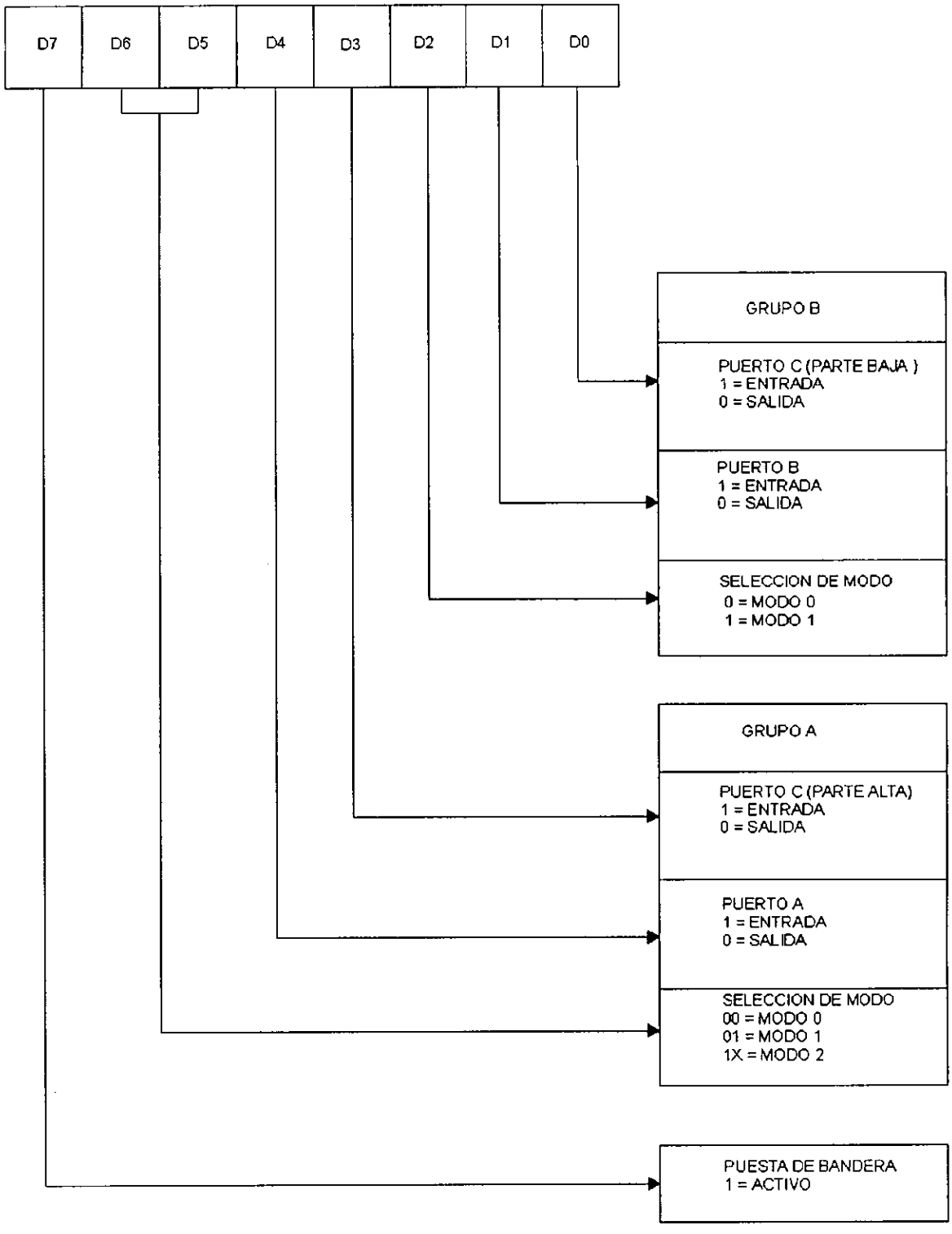


FIGURA 4 ESTRUCTURA DE LA PALABRA DE CONTROL

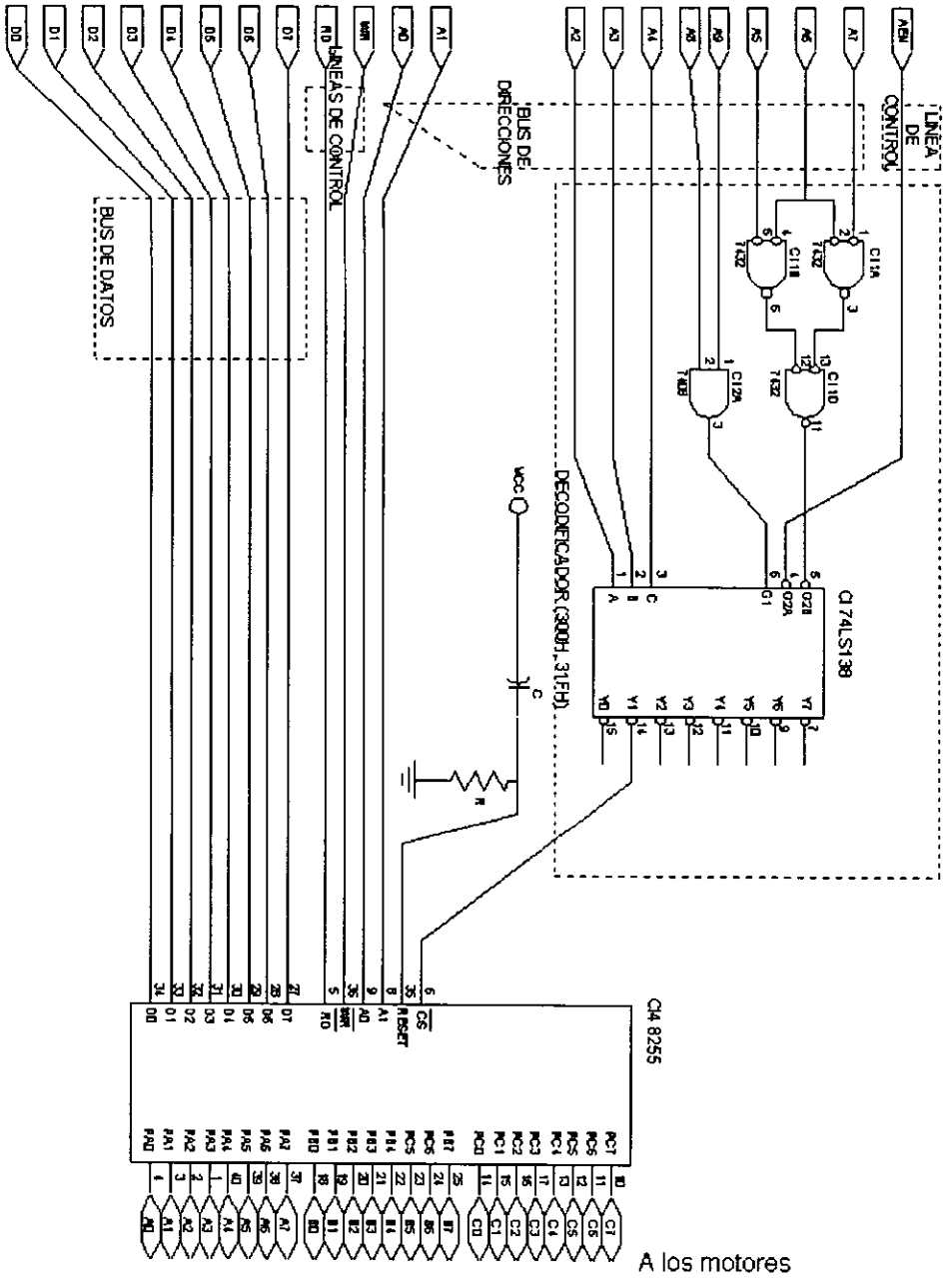


Figura 5 Tarjeta acoplador para el CI 8255A

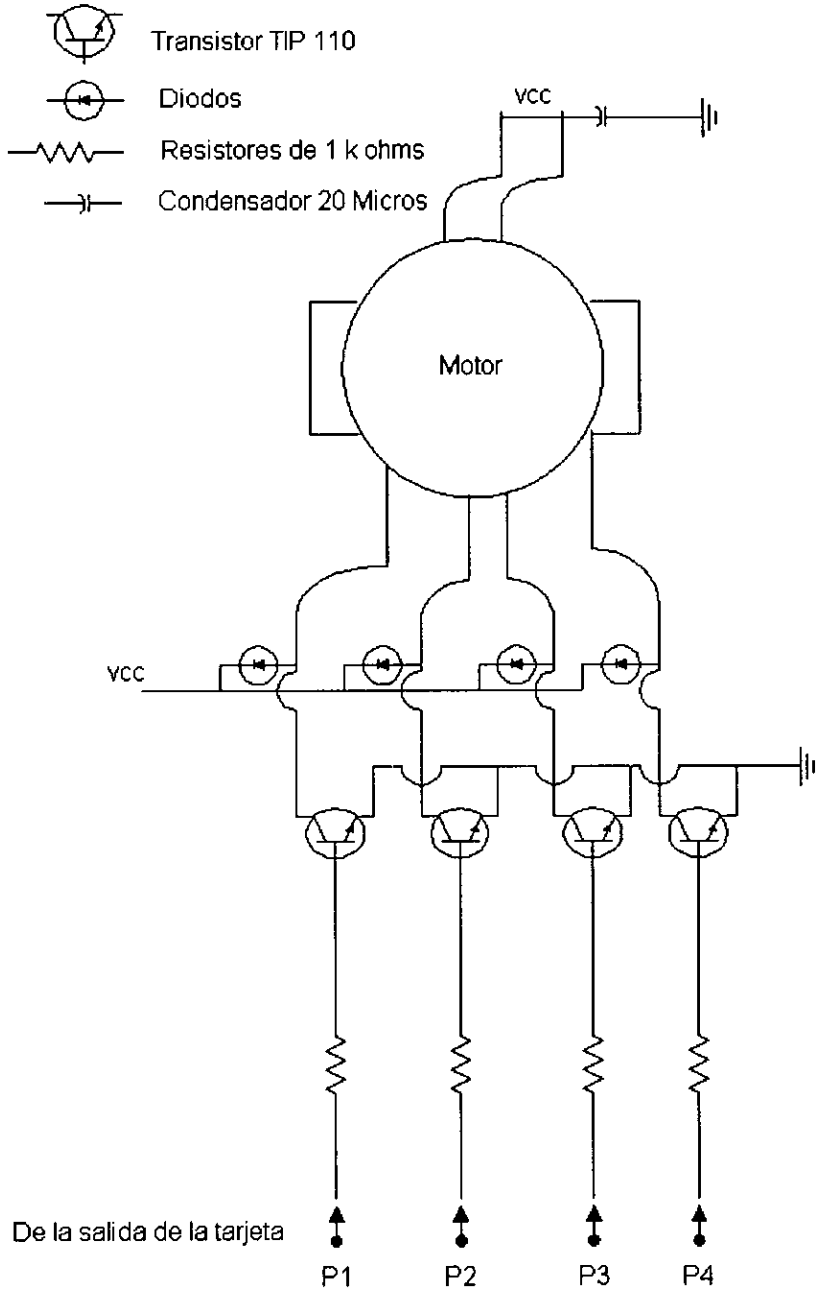


Figura 6 Fase de potencia para activar el motor de 4 fases

APENDICE B

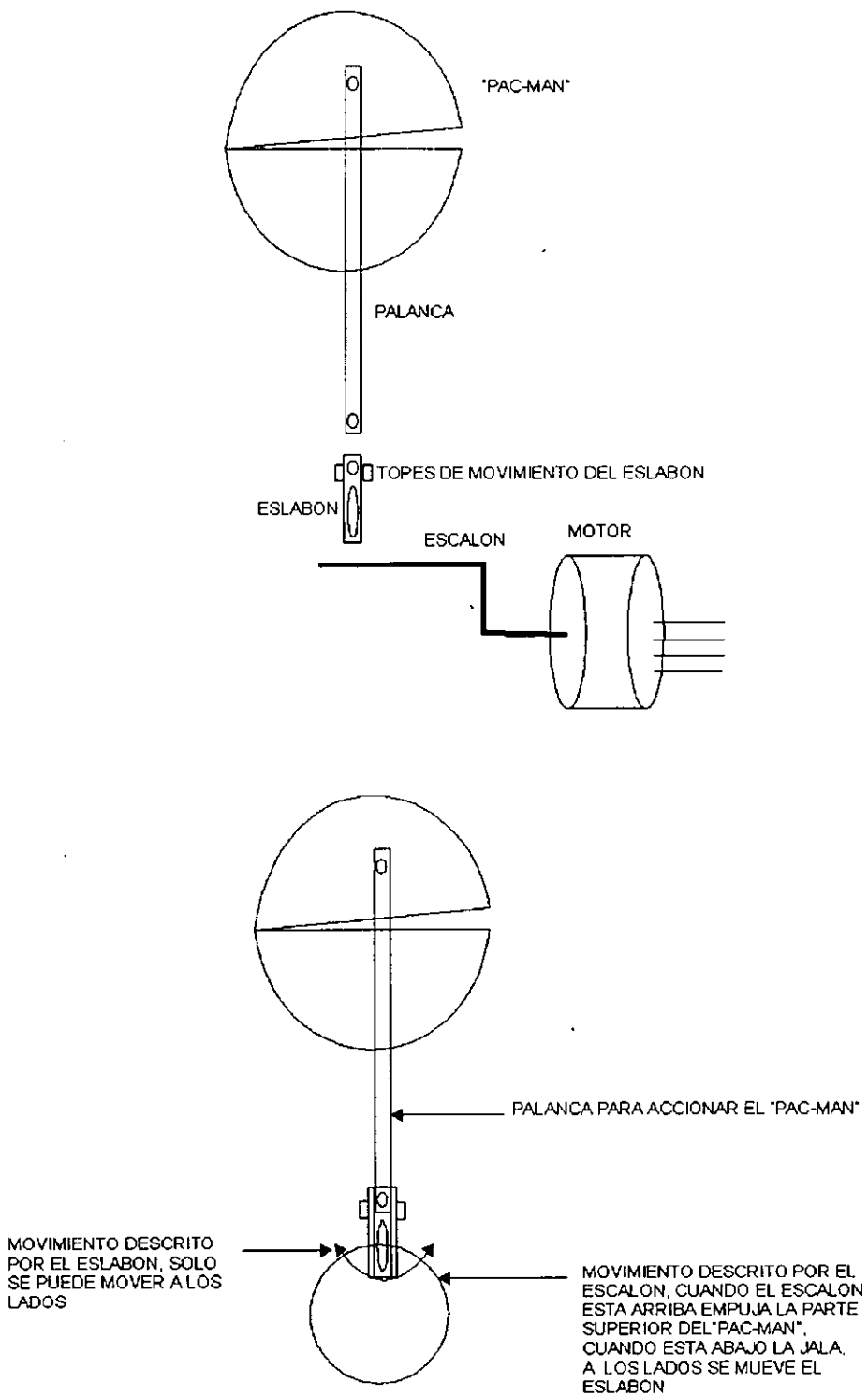


FIGURA 1 DIAGRAMA DE DISEÑO DEL "PAC-MAN"

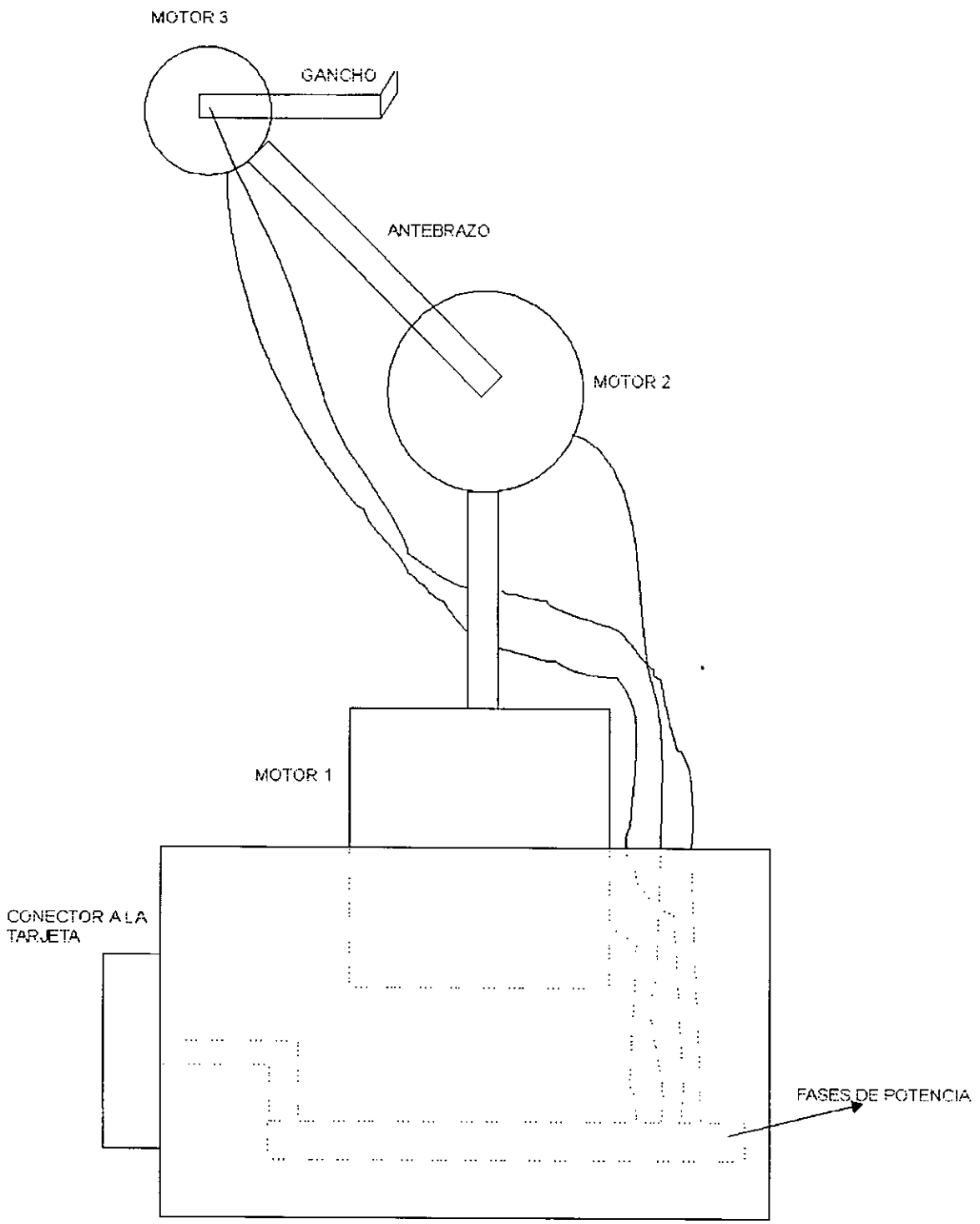


FIGURA 2 DISEÑO DEL BRAZO PARA LA APLICACION PARALELA

APENDICE C

SR

(Synchronizing Resources)

Tipos, variables y asignaciones

- Identificadores son secuencias de caracteres, dígitos y subguiones, el primer caracter debe ser una letra
- Las palabras clave son identificadores especiales, con significado en el lenguaje.
- Literales son valores específicos de tipos diferentes
- Operadores y separadores son palabras clave o caracteres detales como <, >, =

Tipos básicos

SR tiene cinco tipos básicos: booleano, enteros, caracteres, reales y cadenas, representados por las palabras clave: bool, int, real, char y string

Booleanos

El tipo booleano tiene dos literales: *false* y *true*. Los operadores booleanos están representados por las palabras clave: and, or, xor(o exclusivo) y not, cada operador regresa un resultado booleano.

Enteros

Las literale enteras son secuencias de dígitos sin signo. Los números decimales son secuencias de 0 a 9. Los números octales son secuencias de 0 a 7 seguidas por q o Q. Los números hexadecimales son secuencias de 0 a 9, a a f o A a F, seguida de x o X, deben empezar con un dígito.

Reales

Una literal de punto flotante tiene la forma:

parte_entera. Parte_fraccional parte exponente

La parte del exponente empieza con e o E, opcionalmente el exponente y la secuencia de dígitos:

1.23 E-45

Caracteres

Las literales tipo caracter son caracteres sencillos en ASCII encerrados en comillas sencillas:

'a' '\'' '\e' '\33'

Cadenas

Son secuencias de cero o mas caracteres ASCII encerradas en comillas dobles, al igual que el tipo caracter puede contener los caracteres especiales:

\n	Salto de linea
\t	Tabulador
\b	Regreso de espacio
\r	Enter
\'	Comilla sencilla
\\	Diagonal inversa
\ooo	Partrón de bit ooo son dígitos octales
\xhh	Patrón de bits hh dígitos octales
\c	Caracter, c es cualquier caracter
\e	Alerta (campana)
\e	Escape
\v	Tabulador vertical
\f	Forma continua
\"	Comas dobles

Ejemplos:

"", "44", "comillas \" en medio"

Una parte implícita de las cadenas es el número de caracteres que tiene. La declaración de una cadena debe especificar al tamaño máximo de la longitud. El número real de caracteres puede variar desde cero hasta el tamaño máximo especificado. La función predefinida length(s) regresa el número máximo de caracteres en la cadena.

Operadores relacionales y tipos ordenados

Los operadores relacionales comparan sus operandos y regresan un valor booleano que refleja el resultado de esa comparación, se definen con los reales, cadenas y tipos ordenados. De los tipos básicos, booleanos, caracteres y enteros son tipos ordenados, es decir, tiene sentido definir un ordenamiento entre valores entre esos tipos y sus valores sucesivos difieren por una cantidad fija. Reales y cadenas no son considerador ordenados porque los valores adyacentes difieren en una cantidad variable.

El ordenamiento entre valores booleanos es que falso precede a verdadero. Entre valores caracter, el ordenamieto está definido por la representación tratada como enteros sin signo. En valores enteros es por su valor numérico, lo mismo se aplica para valores reales. Las cadenas se ordenan lexicográficamente

usando su representación en caracteres, la cadena vacia es la cadena más pequeña.

Tipo definidos por el usuario

Pueden ser declarados en declaraciones de tipo y referenciados por identificador del tipo. Tales tipos pueden ser declarados anónimamente siplemente definiendo en cualquier punto el tipo requerido

Declaraciones de tipo

La declaración de un tipo introduce un nuevo identificador que es un sinónimo para el tipo declarado:

type type_id = definición

Defincón podría ser un tipo básico declarado opcionalmente precedido por una dimensión, sin dimensión, el nuevo nombre es solo un alias para el nombre abterior, con dimensiones, el nuevo nombre es una alias para un arreglo del tipo definido.

type score_array[0: 200] int
type grados['a': 'e'] int
type direccion = [3] string[40]

El constructor de un arreglo se usa para especificar el valor del arreglo entero. Contiene una lista, encerrada entrea paréntesis, de uno o más elementos separados por comas.

(0, 0, 0) = ([3] 0)
(1, 2, 0, 0, 6) = (1, 2, [3] 0 , 6)
([3] " ") = ("smith", [2] "jones")

Para incializar un arreglo de cadenas de diferentes tamaños se debe asignar valores en forma individual a través de un ciclo.

Los siguientes ejemplos declaran arreglos de 10 x 10 de reales

([10] ([10] 0.0))
([5] ([10] 0.0), [5] ([10] 1.0))

Definen dos arreglos de dos dimensiones, el segundo contiene ceros en sus primeros cinco renglones y unos en los últimos cinco.

Enumeraciones

Cada tipo enumeración es un tipo ordenado, su definición consiste de una lista de uno o más identificadores separados por comas:

enum (id, id, . . .)

Algunos ejemplos podrían ser:

enum (rojo, azúl, amarillo)

Que podría ser usada de la siguiente manera:

type colores = enum(rojo, azúl , amarillo)

Registros

Un tipo registro define una colección de valores de datos, su definición contiene una una lista dde uno o más campos separados por punto y coma:

rec (campo_id; campo_id; . . .)

Cada definición de campo define uno o más nombres de campo del mismo tipo

El tamaño de cada campo debe ser determinado al momento de compilar.

type person=rec(nombre: string[10]; grado: int)

El constructor de un registro se usa para especificar el valor de un registro, contiene el identificador del registro y una lista de una o más expresiones una para cada campo en el tipo registro separadas por comas.

id (expr, expr, . . .)

person("Carlos", 6)

Uniones

Un tipo unión es una colección de diferentes tipos cada uno de los cuales tiene un nombre

union(campo, campo, . . .)

El valor de la unión es el del nombre del campo nombrado, el valor de la unión puede cambiar en el tiempo de ejecución

type person=union(nombre: string[10]; id: int)

Punteros

El tipo puntero define una referencia a un objeto:

ptr type o ptr any

El primer caso define un puntero a un objeto del tipo especificado, el segundo define un puntero a un objeto de cualquier tipo.
Se pueden declarar alias para los punteros de l siguiente forma:

type pint = ptr int
type ppint = ptr ptr int
type pr = ptr rec(i1, i2: int)
type pany = ptr any

Los tipos de registro mutuamente recursivos se pueden definir de la siguiente forma:

type r1 = rec(a: char; p2: ptr r2)
type r2 = rec(b: real; p1: ptr r1)

La declaración de rq usa a r2 antes de que sea declarada.

El puntero genérico, any, puede referenciar un valor de cualquier tipo.

La literal null se usa para indicar que el puntero apunta hacia un objeto no válido. El operador de dirección @ regresa la dirección de la variable, la cual puede ser asignada a un puntero.

Variables y Constantes

La declaración de una variable contiene una lista de una o más definiciones de variables separadas por comas.

var variable1, variable1, . . . , type

Una variable incializada ewspecifica el nombre de la variable, opcionalmente su tipo, y su valor inicial.

variable : type := expr

El nombre de la variable puede tener dos formas:

nom_variable o nom_variable dimensiones

La declaración de una constante contiene una lista de una o más variables inicializadas separadas por comas:

const var_inicializada, var_inicializada, . . .

const N: int := 100, dosN: int := 2*N

const vector[4]: int := (12, 13, 14, 15)

Almacenamiento dinámico

Dos funciones predefinidas permiten el alamacenamiento dinámico:
new (type) y free(expr)

La función new regresa a un puntero a una nueva instancia de almacenamiento para una variable con tipo especificado por el argumento. Si no hay suficiciente espacio para atender la petición de almacenamiento, el programa para. La función free libera el alcenamiento de un objeto almacenado por la función new, apuntado por la expresión argumento.

Los objetos almacenados dinámicamente son referenciados por punteros:

type punto = rec(x, y: real); var a: ptr point

Supóngase que el valor de a ha sido dado por new(point). Entonces a^ es el registro completo al cual a apunta y a^.x y a^.y son sus campos.

Asignaciones

Los siguientes son casos válidos:

j:=k := i * 189

j:= k :0 (i := i+1) * 89

Las asignaciones ampliadas pueden simplificar la escritura de las expresiones:

**:= *:= /:= %:= +:= -:= &:= >>:= <<:= |:= ||:=

Sus significados se derivan de su operador base.

Los operadores de incremento y decremento:

variable++ variable--
++variable --variable

El operador de intercambio intercambia los valores de dos variables, las cuales deben tener el mismo tipo:

variable :=: variable

Conversión de tipos

La conversión de enteros a reales ocurre inplicitamente dentro de las expresiones:

```
var r: real, i: int
r := 3; r:= i*2.34; r := i / 3
```

La conversión implicita de enteros a reales se aplica solo a valores simples no a registros ni a arreglos.

Aunque los enteros son convertidos implicitamente a reales, lo cntrario no sucede.

Control Secuencial

La sentencia skip

Esta sentencia no tiene efecto y termina inmediatamente, su forma:

```
skip
```

La sentencia stop

Termina la ejecución de un programa, tiene dos formas:

```
stop  o stop( expr )
```

Si expr está presente, su valor entero se regresa como es status de salida al comando que inició la ejecución, el valor por omisión es cero.

Procedimientos

Un procedimiento define el código que puede ser invocado, puede recibir parámetros y regresar un valor:

Sin valor de retorno:

```
procedure nombre_proc (lista_formal)
   bloque de código
end
```

Con valor de retorno:

```
procedure nombre_proc (lista_formal)
   returns variable : tipo
   bloque de código
end
```

El nombre del procedimiento puede aparecer después del end.

La lista formal consiste de cero a más parámetros separados por puntos y comas:

```
parámetro1, párametro2, . . .
```

El parámetro define uno o más variables del mismo tipo, separados por comas:

```
variable1, variable2, . . . : tipo
```

en resumen:

```
var1, var2 : tipo; var11, var12 : tipo ; . . .
```

Un procedimiento es ejecutado cuando es llamado, termina cuando ejecuta su última instrucción o un return.

Ejemplo:

```
resource main()
  procedure factorial(x : int)
     var fact := 1
     fa i:=  2 to x -> fact *:= i af
     write ("El factorial de ", x "es", fact)
  end # del procedimiento

  var x : int
  do true ->
     writes( " Teclee un número mayor a 0")
     writes( " o 0 para terminar")
     read (x)
     if x = 0 -> exit
     [ ] x < 0 write (" El número debe ser > 0")
     [ ] x > 0  -> call factorial(x)
  od
end # de main
```

Un procedimiento debe ser declarado antes de ser usado, como en C los prototipos, el encabezado debe ser declarado, no necesariamente el cuerpo del mismo, esto permite recursividad.

Los parámetros pueden ser pasados por copias o por dirección, el valor de regreso es pasado por copia. Los parámetros de valor (val) son copiados hacia la función, los parámetros de resultado (res) y el valor de regreso son copiados hacia afuera de la función y los parámetros variables (var) son copiados en ambos sentidos.

Los parámetros de referencia son pasados por dirección. La clase de parámetros por omisión es val

Pasar un parámetro por copia hacia adentro y hacia afuera (var) tiene el mismo efecto que pasarlo por referencia. El paso por referencia es más eficiente para estructuras de datos grandes.

Los parámetros por referencia no deben ser pasados entre máquinas virtuales (espacio de memoria) debido a que no se puede asegurar el valor de un parámetro de referencia, que es una dirección de memoria.

El tamaño de los parámetros que son arreglos o cadenas puede depender del argumento en la invocación. Esto se denota por el uso de * para cada uno los rangos o para el tamaño de la cadena. El valor de * se vuelve el que sea necesario para adecuar el correspondiente atributo del parámetro real.

Ejemplo:

```
resource main()
  procedure acomodar( var a[1:*] : int)
    fa i := lb(a) to ub(a) -1 ,
      j := i+1 to ub(a) st a[i] > a[j] ->
          a[i]:=: a[j]
    af
  end # de acomodar
  var x[1:20], y[2:30], z['a':'x' ] : int
  call sort(x)
  call sort(y)
  call sort(z)
end # del main
```

El parámetro formal de sort (a) es precedido por var así que los cambios hechos a él por sort se copian de regreso al invocante (en éste caso main)

El límite inferior del arreglo (lb) es siempre 1, el límite superior depende del tamaño del parámetro que se está pasando en casa invocación:

	ub(a)
sort (x)	20
sort (y)	29
sort(z)	26

La sentencia CALL

Un procedimiento es llamado por la sentencia call:

call operación (lista_de_expresiones)

La sentencia call puede ser omitida:

operación (lista_ de_expresiones)

El campo operación es el nombre del procedimiento, lista de expresiones es una lista de cero o más parámetros separados por comas:

expr1, expr2, . . .

Los paréntesis son requeridos

Cuando se ejecuta una sentencia **call** primero se evalúan los argumentos para ver si el número y tipo son adecuados y que corresponden al de los parámetros formales, los argumentos son asignados a los parámetros formales y se ejecuta el procedimiento.

Un procedimiento que regresa un valor, que es parte de una expresión, puede ser llamado también a través de un call.

La sentencia RETURN
transfiere el control, provoca que el procedimiento y la invocación terminen

return

Declaración de operaciones y Proc

La declaración de un procedimiento es una abreviación para la declaración de un proc y de una operación, el requerimiento de que el procedimiento sea declarado antes de ser usado es una limitante - restringe el orden en que los procedimientos deben aparecer dentro del código.

Para permitir la abstracción de datos es necesario poder dividir un procedimiento exportado desde una resource o una global en su parte de especificación (la declaración de la especificación) y su parte de implantación (proc). Las operaciones y los procs son el corazón de los mecanismo de programación concurrente en SR.

La declaración de operaciones permite definir una o más operaciones separadas por comas:

op nombre_op1, nombre_op2, . . .

Cada nombre de operación declara el nombre, el tipo de sus parámetros, cómo son pasados y el valor de regreso, tiene alguna de las siguientes formas:

operac_nom espec_oper

operac_nom espec_oper oper_restricción

La especificación de la operación define los parámetros y el valor de retorno opcional. La forma de la especificación de la operación depende de que haya o no un valor de regreso:

(especifica_lista_parámetros)
(especifica_lista_parámetros) returns var : tipo

La lista de parámetros tiene la misma forma que la lista de parámetros del procedimiento.

La restricción de la operación indica cómo será éste invocada. Las cuatro formas son :

{ call }
{ send }
{call, send }
{ send, call }

ejemplo

op inserta (registro : person)

op diferente(s1, s2 : string[*]) returns lugar :int

op izq(y : int) returns pos : int

Los parámetros y el valor de regreso son opcionales ya que deben ser pasados en la implantación de la operación

op izq (int) returns int

Cada operación es implantada por un proc.[1] La declaración de la operación debe aparecer antes de que el proc sea declarado y antes de que la operación sea invocada. La declaración de un proc tiene una de las dos formas siguientes:

sin valor de retorno

```
proc nombre_oper (lista_formal)
  block
end
```

con valor de retorno:

```
proc nombre_oper(lista_formal) returns result
  block
end
```

La declaración de un proc especifica solo los identificadores que van a ser usados en el cuerpo del proc para referenciar los parámetros formales y para construir el valor de regreso. Los identificadores serán a menudo los mismos que los identificadores correspondientes en la declaración op. El tipo de cada identificador es determinado por el tipo del parámetro correspondiente en la declaración de ña operación, por lo tanto el número de identificadores formales debe ser igual al número de parámetros formales en la declaración de operación correspondiente. Como en los procedimientos no pueden ser anidados.

```
proc insert(ítem)
  if count >= N ->
    write(stderr, "Insert:  sin espacio")
    stop(1)
  fi
  info[++count] := ítem
end

proc differ(s1, s2) returns first_place
  var i:= 1, len:=min(length(s1), length(s2))
  do i <= len( & s1[i] -> i++ od
    if i > len & length(s1) = length(s2)  ->
     firts_place := 0
    []else
     first_place := i
    fi
end differ

proc left(i) returns lft
  lft := (i-2) mod 5 + 1
end
```

Los procs especifican solo identificadores de parámetros no sus tipos o sus métodos de paso. Un proc puede usar identificadores diferentes que aquellos que aparecen en la declaración de la operación correspondiente.

Tipo operación

La declaración tipo operación (optype) crea un nuevo identificador que es una sinónimo para el tipo de operación especificado, puede ser usada en declaraciones de operaciones y **capacidades**. Los tipo operación son a las operaciones lo que la declaración de tipo es a las variables. Declara patrones que serán usados más de una vez

[1] El término "proc" es la abreviación de las palabras procedure y process

optype tipo_op_nom = especificación_operación especificación_restricción

ejemplo

optype intfun = (n: int) returns sum : int

El patrón intfun tiene un entero como parámetro y regresa un valor entero; sin el =

optype intfun (n: int) returns sun: int
optype intfun (int) returns int

Las declaraciones pueden ser optypes en lugar de dar explícitamente especificaciones de operación

op oper_nom1,oper_nom2,...: tipo_op

ejemplo:

op fact : intfun

Declara una operación, fact, cuya especificación está dada por intfun fact debe ser implantada por un proc tal como:

```
proc fact(n) returns prod
  prod := 1
  fa y:=2 to n -> prod*:=i af
end
```

Se usa optype por la abstracción que provee en especificar los parámetros relacionados.

Capacidades de operaciones

Una capacidad de una operación es un puntero a ésta, tales punteros pueden asignados a variables, pasados como parámetros y usados en sentencias de invocación; al invocar una capacidad se tiene el efecto de invocar la operación a la cual apunta.

Una variable o parámetro se define como una capacidad de operación al declarar su tipo en las siguientes formas:

cap especificación_operación
cap nombre_operación
cap nombre_optype

En cada caso la capacidad tiene la parametrización especificada .

Una capacidad de operación puede ser limitada a cualquier operación definida por el usuario, teniendo la misma parametrización. Cuando la parametrización es comparada, solo los nombres de los valores formales y de regreso importan; los identificadores formales y de regreso son ignorados si hay restricciones también deben coincidir. Las capacidades pueden ser comparadas usando el = y el != pero los operadores relacionles (<) no son permitidos, no hay ordenación entre ellos.

Ejemplo

Se tienen las siguientes declaraciones de operaciones:

```
op d(x: int)
op e(var x: int)
optype s = (x: int)
op f(x : real) returns y: real
op y(a : real) returns b: real
```

Se declaran tres variables de capacidad:

```
var x: cap d
var y: cap f
var z: cap s
```

Se pueden usar de la siguiente manera:

```
x:= d # x punta a la operación d
x(387) # invoca a la operación con el argumento 387
if . . . -> y:= f
[ ] else -> y:= g # hace que y apunte a f o g
fi
write(y(4.351)) # se invoca a lo que apunta y
z:= x
 if y=f -> . . .fi # se comparan capacidades
```

Las siguientes sentencias no son válidas

```
x := e # el parámetro de e es una variable, el de x es un valor
d := e # d es una operación no se pueden hacer asignaciones a
ella
z(3, 45) # z requiere de sólo un parámetro
```

El nombre de una operación actúa como una constante, no se le puede asignar, pero su valor puede ser usado para invocar una operación, ser asignadas a variables de capacidad y ser comparadas con otras capacidades.

Ejemplo:

Para calcular el área bajo la curva usando la regla del trapecio:

```
var x:= a
var h := (b-a) / n
area := (f(a) + f(b)) /2
fa  i := 1 to n-1 ->
  x+:= h; area +:= f(x)
af
area *:= h
```

El procedimiento se puede usar de la siguiente manera:

```
procedure fun1(x: real) returns fx: real
  fx := x*x + 2*x + 4
end

procedure fun2(x : real) returns fx : real
  fx := sin(2*x)
end

write (trapecio(0.0, 1.0, 200, fun1))
write (0.0, 3.141592, 1000, fun2))
```

Las variables de capacidad puede tomar 2 valores especiales: null y noop. La invocación de una variable de capacidad que regresa un valor null provoca un error en tiempo de ejecución. En general la invocación de una variable de capacidad cuyo valor es noop no tiene efecto.

Recursos y Globales

Un recurso define un esquema para el cual las instancias pueden ser creadas dinámicamente, también pueden ser destruidas. Una global es esencialmente una instancia de un recurso sencillo sin parámetros creada automáticamente.

Un recurso puede ser visto como un tipo abstracto de datos y consiste de una parte de **especificación**, la cual indica la interface del recurso y una parte cuerpo, la cual contiene el código que implanta el objeto abstracto.

Una global es una colección de objetos compartidos por recursos y otras globales en el mismo espacio de direcciones (máquina virtual). Las globales pueden ser usadas de diversas maneras. En su forma más simple contiene declaración de tipos, constantes y variables que son usadas a través del programa. En esta caso una global contiene solo una parte de la especificación. En su modo más complejo una global contiene también operaciones y código asociado para generar una librería. En todo caso la global tiene la parte de la especificación y cuerpo del código.

En todos los casos la forma de la global es un poco similar a la del recurso, sin embargo existen 2 diferencias clave que demuestran la utilidad de las globales: la primera es que las globales permiten que las variables sean compartidas por todas las instancias de los recursos que la importan; los recursos permiten que las variables sean compartidas solo por procs dentro de cada instancia del recurso. Segundo, los proc declarados en el spec de una global pueden ser referenciadas fuera de la global directamente a través del nombre de la global, los procs declarados en el spec de un recurso deben ser referenciados fuera del recurso indirectamente a través de una capacidad del recurso para la instancia del recurso. Estas diferencias afectan la manera de escribir los programas y su desempeño.

El número de instancias de recursos en un programa en SR puede crecer o disminuir durante la ejecución. Un programa en SR puede contener múltiples instancias de una global en diferentes máquinas virtuales.

Especificación de Recursos y Cuerpos

Cada recurso está compuesto de dos partes: la **spec** la cual especifica la interface del recurso y el **cuerpo** el cual contiene el código que implanta el recurso:

```
resource resource_name
  imports
  constan, types, operation declarations
  body resource_name(parámetros)
       imports
       declaraciones, sentencias, procs
       final code
end resource_name
```

Los parámetros y todas la partes en la spec y el cuerpo son opcionales, así como el nombre del recurso después del end.

Los objetos declarados en la spec de un recurso son implícitamente exportados y aún son visibles a otros recursos. La cláusula **imports** se usa para obtener acceso a objetos exportados por otros componentes. La spec para un recurso también da la parametrización para instancias de ese recurso, esos parámetros deben ser pasados por valor. La spec de un recurso no puede contener declaraciones de variables, sentencias, arreglos de operaciones o semáforos, debe tener tamaños constantes.

El cuerpo de un recurso da la implantación del recurso. El cuerpo contiene procs, los cuales implantan acciones y declaraciones las cuales introducen objetos compartidos por todos los componentes dentro del cuerpo, puede contener opcionalmente código inicial y/o código final. El código inicial consiste de las sentencias ejecutables en el cuerpo, es ejecutado cuando se crea el recurso, el código final, que es un bloque de código dentro del **final** y el **end** se ejecuta cuando el recurso se destruye.

Los componentes del cuerpo pueden ser declarados en cualquier orden. Esto es útil para agrupar variables y procs que las usan.

Ejemplo

```
resource stack
  typeresult =
      enum(OK, OVERFLOW, UNDERFLOW)
  op push(item: int) returns r: result
  op pop(res item : item) returns r: result
  body  stack(size: int)
  var store[1: size]: int, top : int := 0
  proc push(item) returns r
   if top < size ->
    store[++top]:= item; r:=OK
  [] top = size -> r:= OVERFLOW
  fi
end
proc pop(item) returns r
  if top > 0 -> item:=store[top--]; r:=OK
  [] top = 0 -> r:=UNDERFLOW
  fi
end
end stack
```

La spec declara dos operaciones push y pop y un tipo result; éstos son visibles fuera del recurso. El cuerpo del stack contiene la declaración de las variables usadas al implantar el stack y los procs que implantan las dos operaciones. El número de elementos en store depende del parámetro del recurso.

Size

El spec y el cuerpo de un recurso pueden ser separados. Las partes pueden aparecer en el mismo archivo o en archivos separados. En cualquier caso el cuerpo debe ser compilado después de su spec y debe ser recompilado siempre que la spec cambie.

Imports

Cuando un recurso quiere usar otro debe importarlo. Una cláusula imports especifica uno o más recursos o globales separados por comas:

```
imports name, name, . . .
```

puede aparecer en cualquier parte que se pueda poner una declaración, es decir, en el spec de un recurso o en el código de un bloque. La cláusula imports da acceso a los objetos en los recursos o globales nombrados, esos objetos son visibles desde el punto de la cláusula imports hasta el final del bloque que está importando

```
resource stack_user()
  imports stack
  var x stack.result
. . .
end
```

Creando y destruyendo instancias de un recurso

Para crear una instancia de un recurso se usa la expresión create

```
create resource_name(argumentos)
```

Esta regresa una capacidad para un recurso, la cual actúa como un puntero a la instancia del recurso nombrado. Esta capacidad es usada subsecuentemente para invocar operaciones en la instancia del recurso o para destruir la instancia.

```
cap_var := create resource_name(argumentos)
```

cap_var debe tener el tipo cap resource_name

La ejecución de una cláusula create crea una instancia del recurso nombrado se asignan los valores a sus parámetros y el código inicial se ejecuta. La expresión create termina y regresa una capacidad al recurso cuando el código inicial en la nueva instancia se ejecuta

```
resource stack_user()
  imports stack
  var x: stack.result
  var s1, s2: cap stack
  var y: int
  s1:= create stack(10)
  s2:= create stack(20)
. . .
  s1.push(4)
  s1.push(37)
```

```
  s2.push(98)
  if (x:=s1.pop(y)) != OK -> . . . fi
  if (x:=s2.pop(y)) != OK -> . . . fi
. . .
end
```

Las operaciones y parámetros declarados en el spec y las variables declaradas en el cuerpo son específicas de la instancia. En el ejemplo cada instancia de stack tiene sus propias copias de las operaciones push y pop, su propio valor de size y su propia copia de top y store. Las operaciones se referencían fuera del recurso a través de la variables de capacidad, por ejemplo s1.push (dentro del recurso las operaciones pueden ser referenciadas directamente). Por el contrario las constantes y tipos declarados en el spec son asociados con el recurso no con cada instancia así result es asociado con el recurso. Este es, por lo tanto, referenciado como stack.result (o simplemente result si el nombre es único).

La ejecución de un programa empieza con la invocación implícita de una instancia del recurso principal (main) del programa.

Ese código inicial del recurso es ejecutado y puede crear otros recursos. El recurso principal no puede ser importado por ningún otro recurso en ese programa, éste se especifica cuando el programa se liga.

La sentencia destroy destruye una instancia del recurso:

destroy cap_var

Cuando una instancia de un recurso es destruida su código final, final, se ejecuta. La ejecución del destroy provoca un run time error si la instancia ya ha sido destruida.

Cuando un programa SR termina (cuando todos los procesos en el programa han terminado o cuando una sentencia stop se ejecuta) con un estado de salida cero, la instancia inicialmente creada del recurso main es implícitamente destruida. Aunque cualquier código final será ejecutado.

Variables con capacidad de recursos

Las expresiones create regresan éste tipo de variables y son usadas con las sentencias destroy, pueden ser comparadas o asignadas. La nomenclatura de un recurso es determinada por las operaciones que ésta declara en su spec. Dos recursos son comparables por su nomenclatura si declaran el mismo número de operaciones compatibles por la nomenclatura, en el mismo orden.

```
resource st
  type
    status = enum( NORMAL, OVER, UNDER )
  op add(item : int) returns r : status
  op drop(res : int ) returns r : status
  body st() separate
```

Es comparable con el recurso stack. Lo único que importa es el número de operaciones, el orden en el cual fueron declaradas, el tipo de sus parámetros y valores de regreso, por lo tanto el siguiente fragmento de código es válido:

```
var s1, s2: cap stack, s3: cap st
s1 := create stack(10)
s3 := s1
```

s3 y s2 apuntan a la misma instancia.

```
S3.add(5)
```

Invoca la operación push del recurso stack

También se pueden usar en asignaciones a o de capacidades de operaciones:

```
var c: cap (item: int) returns r: stack.result
c := s1.push()
```

Las invocaciones de c son invocaciones de s1.push()

Las variables con capacidad de recursos se les puede asignar el valor null y noop. La invocación de una variable cuyo valor es null provoca un error en tiempo de ejecución, si su valor es noop no tiene efecto.

El código inicial y final

El código inicial aparece al principio en el código del recurso, está al mismo nivel que las variables de los procs y recursos.

El código final aparece como un bloque de código sencillo. Es introducido por la palabra final y terminado por la palabra clave end. Puede aparecer en cualquier lugar en el cuerpo del recurso, pero es puesto usualmente el final. Se usa para limpiar antes de que la instancia del recurso desaparezca.

Extendiendo recursos y recursos abstractos

Un recurso puede extender a otros. La cláusula extend puede aparecer en la spec de un recurso, contiene uno o más nombres de recursos separados por comas:

extend recurso1, recurso2, . . .

El efecto es pasar todas las declaraciones del recurso extendido incluyendo las cláusulas import y extend tal como si fueran declaradas en el recurso que contiene la cláusula extend, implícitamente importa el recurso.

Si un recurso extiende a más de uno, los recursos extendidos son heredados en el orden que estén nombrados en la cláusula extend. El efecto de extend es acumulativo: un recurso hereda los objetos declarados en los recursos que él extiende.

```
resource newstack
  extend stack
  op count() returns r : int
  body newstack( size: int) separate
```

newstack declara tres operaciones y un tipo result, push y pop de stack y count declarada aquí. El cuerpo de newstack debe proveer código que implante las tres operaciones, el código de stack no se incluye automáticamente en el cuerpo de newstack. SR no soporta la herencia de código como en lenguajes orientados a objetos, sólo soporta herencia en las declaraciones

El recurso abstracto es un recurso que no tiene cuerpo:

```
resource nombre
  imports
  constant, tipo operaciones
end
```

Recursos que se importan mutuamente

El spec de un recurso se importa cuando se encuentra la primera vez importaciones posteriores se ignoran; esto previene la recursión infinita. Sin embargo, las declarciones precedentes a las sentencias de mutua importación ya en sido procesadas cuando ésta regla tiene efecto, así que quedan disponibles para ambos recursos.

Un recurso puede referenciar a un objeto en otro spec de un recurso en cualquier ocasión después de haberlo importado. Además el segundo recurso puede referenciar declaraciones en la spec del primer recurso en tanto que éstas aparezcan antes de la cláusula imports que importa al segundo recurso.

```
resource A
  const u:=100
  imports B
  const v:= B.x + 10
  op foo(c: cap B)
end
resource B
  const x:=200
  imports a
  const y:=A.u + 20
end
```

El valor de A.u es 210 y el valor de B.y es 120. El parámetro en la operación foo es una capacidad para recurso B, si A se cambia por:

```
resource A
  const u:=100
  import B
  const v:=B.y + 10
end
```

El problema que se presenta es el de un ciclo; v depende de y, la cual a su vez depende de u. La regla a considerar es una spec o debe depender de nada en otra spec que aparece después de la cláusula que la importa.

La forma simple de las globales

La forma más simple de una global es cuando se usa para declarar constantes, tipos y variables que serán compartidas por recursos u otras globales. En éste caso la global sólo consiste del spec

```
global nombre
  imports
  declaraciones
end
```

Los objetos declarados en el spec de una global son visibles a los componentes que la importan

```
global caracteres
  const TAB:='\t'
  const CR:='\r'
end

global node
  type node = (value: char; link : ptr node)
  type head = ptr node
  type tail = ptr node
end
```

```
global matriz
  const N:=20
  var m[N, N]: int ([N] ([N] 0))
end
```

Estas tres globales podrían usarse en un recurso como sigue:

```
resource foo()
  import caracteres, node, matriz
  var x: node.node
...
  if x.value = caracteres.TAB -> . . .fi
...
  matriz.m[3,4] :=: matriz.m[4,3]
...
end
```

node, TAB y m son referenciados en foo usando sus nombres calificados. Si esos nombres fueran únicos y no entran en conflicto con los nombres de las funciones predefinidas, se puede usar el nombre sin el calificar.

La forma general de las globales

Las globales también tienen cuerpos. Esta forma de global es usada cuando la inicialización de variables declaradas en el spec es más complicada de lo que puede ser especificado en el spec. Un cuerpo puede ser usado también para implantar operaciones compartidas. Por ejemplo un procedimiento compartido se implanta poniendo sus declaración de operación en la spec de la global y su declaración de operación de proc en el cuerpo (una declaración de un procedimiento no puede aparecer en la spec por que contiene código).

La forma de la global es:

```
global nombre
  import
  declaraciones
body nombre
  import
  declaraciones, sentencias, procs
  final
end nombre
```

El cuerpo de una global debe ser compilado antes que la global sea importada por un recurso u otra global.

A lo más un instancia de una global será creada por máquina virtual. Una instancia de una global se crea implícitamente cuando se en encuentra el primer import de esa global durante la ejecución. Esta primera import puede aparecer dentro de la especificación de un recurso, en cuyo caso la global se crea durante la creación de la primera instancia del recurso. Debido a que una global puede importar a otra, la creación recursiva puede ser requerida.

Como las globales son creadas en el punto donde se importan, las variables exportadas por las globales existen y son inicializadas antes de que se usen por el código importador.

Las globales importadas directamente o indirectamente por el recurso principal se destruyen cuando el programa termina. Si una global A importa una global B, entonces A es finalizada antes que B si el código final de A usa objetos de B.

```
global diagonal
  const N:=20
  var a[N, N]: real := ([N] ([N] 0.0))
body diagonal
  fa i:=1 to N -> a[i, i]:=i af
end
global screen
  op refresh(), move_to(x, y : int)
  op pop(), down(), left(), right()
  op write(x: string[])
body screen
  proc refresh()
...
  end
  proc move_to(x, y)
...
  end
  var where_x :0, where_y :0
  refresh()
  move_to(where_x, where_y)
end screen
```

Todos los objetos declarados dentro del cuerpo de screen son privados a ella. Los únicos objetos visibles a los objetos que importen a screen son los declarados en la spec.

Screen podría ser escrita como un recurso, sin embargo esto requeriría (1) crear una instancia del recurso y (2) pasar una capacidad para cada recurso que desee invocar sus operaciones.

Entrada / salida y operaciones externas

En SR la entrada y salida es soportada por un tipo adicional de datos: file. Una variable tipo file contiene un descriptor para un archivo UNIX. Existen 5 literales de archivo predefinidas,

para las cuales hay palabras reservadas tres de ellas se usan para accesar archivos

- stdin
- stdout
- stderr

las otras son null y noop

Además existen 2 enumeraciones predefinidas:

```
type
modo=enum(READ, WRITE, READWRITE)
type
busqueda=
enum(ABSOUTE, RELATIVE, EXTEND)
```

éstos tipos se usan como argumentos para las operaciones open y seek. Existe una constante predefinida:

```
const EOF:=-1
```

La archivos se crean y se abren por medio de una función predefinida open. Los archivos pueden ser **flushed**, cerrados y borrados usando las funciones predefinidas flush, close y remove

```
var f: file
f:= open("foo", read)
if f = null
write("Error, no se puede abrir el archivó")
stop(1)
fi
```

I/O simple

Se usan las funciones read, write y writes
read(stdin, x) es lo mismo que read(x)

El primer argumento de cada función puede especificar el archivo al cual la función se aplica, si se omite se usa stdin para leer y stdout para escribir

```
var m[n, n]:=([n] ([n] 0))
do
  r:= read(f, y, j)
  if f = EOF -> exit
  [] r=0 or r=1 ->
    write (stderr, "archivo vacío")
  stop(1)
fi
if y>1 or i>n or j<j or j>n ->
 write (stderr, "Coordenadas fuera de rango")
 stop(1)
fi
m[i, j] := m[j, y] := 1
od
```

I/O con formato

Printf y scanf proveen un control mayor sobre el formato de salida y entrada. Sprintf y sscanf son similares a los primeros excepto que la salida es puesta o la entrada viene de una cadena.

```
printf(f, fmt, x1, x2, ...)
```

x1, x2, ... son los valores que serán escritos en el archivo f de acuerdo al formato fmt, si se omite f se usa stdout fmt es muy parecido al formato en C

```
printf("a[%d] es %d \n", a[i])
```

```
if sscanf(s, "%2d %3d", a, b) != 2 ->
write (stderr, "formato equivocado")
stop(1)
fi
```

Se usa sscanf para dividir una cadena en 2 variables enteras, si s es "12345" a = 12 y b = 345. El valor de regreso de sscanf es el número de conversiones exitosas.

I/O de carácter

Get y put tratan la entrada y salida como flujos de caracteres sin interpretación

```
get (f: file; res, str: string[*]) returns int
```

Lee caracteres del archivo f, stdin si f se omite y los almacena en str. Si el archivo de entrada contiene al menos maxlength(str) caracteres, esos son los que se leen, get regresa el número de caracteres que fueron leídos y pone la longitud de str a ése valor.

```
put(f: file, str: string[*])
```

Escribe length(str) caracteres de str al archivo f a la stdout si se omite f.

Acceso aleatorio

Seek y where son las funciones que se usan en el acceso aleatorio de datos dentro del archivo. Ambos toman como argumento el nombre del archivo, que debe estar abierto. Un puntero de lectura/escritura es asociado con cada archivo, éste conserva el desplazamiento dentro del mismo de donde se aplicará la siguiente operación de I/O, inicialmente está al principio del archivo y tiene un valor de cero. Al invocar seek nueve el puntero; where regresa el valor actual de ése puntero.

```
seek(f: file; t: seektype; offset: int)
returns pos: int
```

El efecto de invocar seek depende del tipo de búsqueda t, como sigue:

t	efecto en el puntero e/s
ABSOLUTE	se pone en offset
RELATIVE	se incrementa en offset
EXTEND f	final del archivo más offset

El valor de regreso es el nuevo valor del puntero se entrada salida. Cuando t es RELATIVE o EXTEND el valor de offset puede ser negativo de acuerdo a las posición resultante del archivo

```
where(f: file) returns pos: int
```

Regresa la posición actual del puntero entrada/ salida

```
var f: file, e: int
f:= open("foo", READWRITE)
e:= seek(f, EXTEND, 0) #tamaño del archivo
fa y:=0 to by 2 ->
seek(f, ABSOLUTE, i);
put(f, "*")
af
close(f)
```

Determina el número de caracteres en el archivo. El ciclo sobrescribe * en el archivo empezando en la primera posición.

Argumentos en línea

Numargs regresa el número de argumentos no contando el nombre del comando. Getargs lee los argumentos en línea, especificados posicionalmente por el primer parámetro en su segundo parámetro (resultado). El tipo del segundo parámetro de getarg puede ser int, bool, char, real, string, arreglo de caracteres, puntero o enum (aunque las literales no son reconocidas)

```
var pn: string[40]
if getarg(1, pn) = EOF ->
  write(stderr, "Sintaxis: a.out archivo [inicio]")
  stop(1)
fi
var sn:= 1
getarg(2, sn)
```

En el primer caso getarg intenta leer el nombre del archivo de la línea de comando. El segundo intenta leer un entero de la línea de comando, si el segundo argumento no está presente o no es un entero sn no cambia y la ejecución continua.

Operaciones externas

Para permitir el acceso a operaciones escritas en otros lenguajes, SR soporta operaciones externas para funciones compatibles con la secuencia de llamado de C.

Una operación externa se declara como una operación regular, usando la palabra external. Las operaciones externas pueden ser declaradas sólo a nivel de recurso o global, no dentro de un proceso, se les invoca como una operación regular y puede regresar un valor.

```
external hostname(res s: string[*]; name: int)
. . .
var hname: string[31]
hostname(hname, maxlength(hname))
```

Concurrencia

Una declaración de operación define una interface de comunicación, un proc define cómo las invocaciones de esa operación van a ser atendidas.

Operaciones, procs y calls son los tres mecanismos base para la programación concurrente en SR, junto con las sentencias send e input.

Cuando un proc es llamado (call) el llamador espera que el proc regresa. SR también tiene sentencia send, la cual puede ser usada para crear un flujo a una nueva instancia de un proc. Mientras que call es síncrono (el llamador espera) send es asíncrono (el emisor continua). Si un proceso invoca a un proc el enviar un send a la operación correspondiente, un nuevo proceso se arranca para ejecutar el cuero del proc y entonces el proceso emisor y el nuevo proceso ejecutan **concurrentemente.**

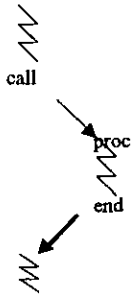
Los procesos en un programa concurrente necesitan ser capaces de comunicarse y sincronizarse. Procesos en el mismo recurso

pueden compartir operaciones y variables declaradas en ese recurso. Los procesos en el mismo espacio de direcciones pueden compartir variables y operaciones exportadas por las globales. Los procesos comunicarse también por medio de la sentencia de entrada (in), la cual atiende a una o más operaciones. Un proceso que ejecuta una sentencia de entrada espera hasta que una de esas operaciones es invocada, atiende una invocación, opcionalmente regresa un valor y entonces continua. La comunicación puede ser síncrona (call) asíncrona (send). El call produce una comunicación de dos vías además de la sincronización (*rendezvous*) entre el llamador y el proceso que ejecuta la sentencia de entrada (in). Un send produce una comunicación de una vía es decir *paso de mensajes asincrono.*

En resumen los mecanismos básicos para la programación concurrente son operaciones y diferentes forma de invocarlas y servirlas. Las operaciones pueden ser invocadas en forma síncrona (call) o asíncrona (send) y pueden ser atendidas por un proc o por una sentencia de entrada in. Se producen las siguientes combinaciones:

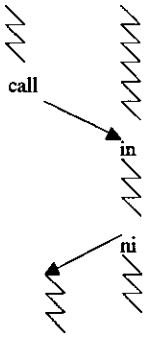
Invocación	servicio	efecto
a) call	proc	llamado a procedimiento (posiblemente remoto)
b) call	in	rendezvous
c) send	proc	creación dinámica de procesos
d) send	in	paso asíncrono de mensajes

a)



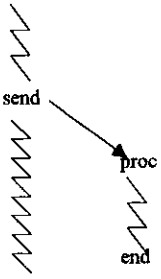
Llamado a procesos remotos

b)



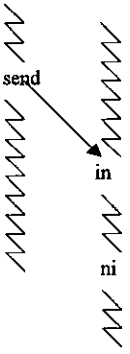
Rendezvous

c)



Creación dinámica de procesos

d)



Paso asíncrono de mensajes

Ejecución concurrente

Un recurso o una global, además de contener procs y procedures, pueden contener processes, la forma simple de un process:

```
process process_id
  block
end
```

La declaración puede contener una lista de una o más cuantificadores para especificar múltiples instancias del mismo proceso, así que una familia de procesos tiene la forma:

```
process process_id (cuant1, cuant2, . . .)
  bloque
end
```

Los process en un recurso o global se crean cuando el recurso o global se crea, los process se crean cuando sus declaraciones se encuentran en la ejecución del código inicial del recurso o global. Para declaraciones de process que contienen cuantificadores una instancia del process se crea para cada combinación de valores de las variables índices. Cada instancia del process tiene acceso a los valores asociados de las variables índice, son argumentos pasados implícitamente a las instancias:

```
resource foo()
  var x:=0
  process p1
    x+:=3
  end
  var a[10]: int
  fa i:=1 to 10 -> a[i]:=i af
  process p2
    x+:=4
  end
end
```

Este proceso contiene 2 process p1 y p2, cuando foo se crea, primero se inicializa x, entonces p1 se crea, el arreglo a se inicializa y finalmente se crea el process p2. Los process se ejecutan al mismo tiempo, al menos conceptualmente, en el ejemplo cada process accesa sin ninguna restricción la variable x. Las variables del recurso no son automáticamente protegidas de accesos concurrentes, la exclusión mutua debe ser programada explícitamente; el orden en el cual los process ejecutan es no determinístico, así si hubiera salido a pantalla el valor de x no se podría anticipar cuál sería el valor de x.

```
resource mult()
  const N:=20
  var a[N, N], b[N, N], c[N, N]: real
. . .
  process multiply(i:=1 to N, j:=1 to N)
    var inner_prod := 0.0
    fa k:=1 to N ->
      inner_prod +:= a[i, k] * b[k, j]
    af
    c[i, j] := inner_prod
  end
  final
  # imprime valores de c. .
  end
end
```

Se emplea una familia de process de N por N. Cada instancia de multiply puede determinar su propia identidad a través de i y j, éstas son diferentes para cada instancia.

Un programa con múltiples process termina cuando todos han terminado, ocurre un dead lock o se ejecuta la sentencia stop. Cualquier código final en el recurso principal se ejecuta y las globales importadas se destruyen.

La forma no abreviada de los process.

La forma abreviada es útil cuando el número de instancias a crear es conocido y cuando los process van a ser creados al mismo tiempo que una instancia de un recurso o global. En ocasiones los process necesitan ser creados de acuerdo a como el programa ejecute. Para entender el mecanismo de creación de los process, es necesario examinar las partes que constituyen la forma abreviada.

La declaración de un process es realmente la abreviación para la declaración de una operación, un proc, y una invocación send. Un send no espera a que el proc invocado regrese algún resultado, el send termina inmediatamente de pasar los argumentos al proc, un nuevo proceso se crea para ejecutar el proc, éste ejecuta en paralelo con el process que ejecutó el send.

```
resource foo()
  var x:=0
  op p1() {send}
  send p1() (1)
  proc p1()
    x+:=3
  end
  var a[10]: int
  fa i:=1 to 10 -> a[i]:=i af
  op p2() {send}
  send p2()
  proc p2()
  x+:=4

end
```

p1 y p2 son declaradas como operaciones y su código es escrito como procs. También el código inicial de foo contiene sends explícitos (1) para crear una instancia de cada process.

La declaración de las operaciones p1 y p2 incluyen lo que se llama una restricción de operación:

op p1() {send}

Esto especifica que p1 puede ser invocado solo por sentencias send. La declaración de una operación puede incluir ésta restricción de operación si el programador desea especificar la forma en la cual la operación puede ser invocada. Las dos restricciones son:

{call} y {send}

Una operación que resulta de una declaración de un process tiene la restricción send. Una operación que resulta de una declaración de un procedimiento tiene la restricción call; aunque los procedimientos pueden ser invocados solo por sentencias call. Si una declaración de operación no especifica la restricción puede ser invocada tanto por call y send, esto puede ser especificado explícitamente:

{call, send} o {send, call}

El recurso mult visto anteriormente puede ser escrito de la siguiente manera (sin la abreviación de procesos)

```
op multiply(i, j: int) {send}
  fa i:=1 to N, j:=1 to N ->
  send multiply(i, j)
  af
  proc multiply(i, j)
  # sin cambios
end
```

El process multiply se reemplaza por una operación y un proc, la operación tiene dos argumentos y una restricción send. El ciclo for all en el código inicial crea una instancia del proc para cada par de valores de las variables índice. Se puede apreciar que es mas simple usar la abreviación process, siempre que sea posible:

```
resource compressor
  op compress(nombre: string[*])
  body compressor()
  prod compress(nombre)
...
end
```

Se declara la operación compress en el spec. Una instancia del recurso puede ser creada:

```
var c: cap compressor
c:= create compressor()
```

por lo tanto la operación compress puede ser invocada fuera del recurso como sigue:

```
send c.compress("data1")
send c.compress("data2")
```

Cada invocación de compress provoca que se cree un nuevo process dentro de compressor. El invocante no espera a que los process terminen antes de continuar ejecutando. La declaración process no puede ser usada en casos como éste por que la invocación provoca que los process sean creados fuera del recurso. Instancias adicionales del proc pueden ser creadas dinámicamente por medio de invocaciones send explícitos.

```
Process foo(id:=1 to )
...
end

send foo(17)
```

Invocación concurrente

La invocación de la sentencia concurrente produce otro mecanismo para crear process dinámicamente. Consiste de una o más comandos concurrentes separados por los delimitadores //:

co comando1// comando2// . . . oc

Cada comando consiste de una invocación y opcionalmente un bloque de código de postproceso:

invocación o invocación -> bloque

La invocación que es parte de un comando concurrente es una invocación call, una invocación send o una asignación simple que llama a una función definida por el usuario.

Invocación sin código de postprocesamiento

co p(3) // q() // a:= r(x, y) oc

Consiste de tres invocaciones p, q y r, r asigna su valor a. La invocación de una sentencia concurrente empieza todas las invocaciones en paralelo, cuando no hay código de postproceso, termina cuando todas las invocaciones terminan.

Ejemplo:

Para calcular la suma parcial de enteros de 1 hasta el argumento en línea n. El programa incializa sum[1: n] así que sum[i] es igual a i. Cuando el ciclo do termina, cada sum[i] es la suma de los enteros desde 1 hasta i. Se usa lo que se llama algoritmo paralelo prefijo: empieza con una distancia= 1, entonces suma sum[i-d] a sum[i] en paralelo, (para todas las i mayores que d), se duplica d y repite hasta que d sea mayor a n. Se usa un arreglo temporal para almacenar una copia de sum y evitar interferencia entre procesos (update).

```
resource suma_parcial()
  var d:=1, n: int; getarg(1,n)
  var sum[n], old[n]: int
  procedure save(i: int)
    old[i]:= sum[i]
  end
  procedure update(i: int)
    if i>d -> sum[i]+:= old[i-d] fi
  end
  fa i:=1 to n -> sum[i]:= i af
  do d<n ->
    co (i:=1 to n ) save(i) oc
    co (y:=1 to n ) update(i) oc
    d+:= d
  od
  fa i:=1 to n -> write(i, sum[i]) af
end
```

Invocación concurrente con código de postproceso

Como en el caso anterior, todas las invocaciones empiezan en paralelo. Entonces, de acuerdo a como cada invocación termine, el correspondiente bloque de postproceso se ejecuta, si hay. Los bloques de postproceso son ejecutados uno a la vez, no son ejecutados concurrentemente así se pueden cambiar variables sin requerir exclusión mutua. La ejecución co termina cuando todos los bloques de postproceso terminan o cuando algún bloque de postproceso ejecuta una sentencia exit

```
cnt := 0
co (i:=1 to n st a[i]!=0 ) p(i) ->
cnt++; write(cnt, y)
oc
```

Se usa un cuantificador para invocar p, uno para cada valor de i, tal que a[i] no es cero. El bloque de postproceso cuenta el número de tal invocación en cuanto termina e imprime sus índices. El uso de la variable i es legal debido a que el alcance de la variable cuantificadora se extiende hasta el fin del bloque de postproceso, debido a que el bloque de postproceso se ejecuta una vez a la vez, la actualización de cnt no necesita ser protegida

```
co (i:=1 to 4) fd[i].read(arguments) ->
wich_one:= 1; exit
oc
```

Cuando cualquiera de la invocaciones termina, el código de postproceso guarda en cuál y sale sin esperar por las otras invocaciones.

Si un bloque de postproceso sale antes de que terminen todas las invocaciones, las restantes no serán canceladas, ellas seguirán siendo atendidas, pero el invocante no esperará por ellas.

Semáforos

Si s es un semáforo, V(s) incrementa el valor de s y P(s) detiene a su llamador hasta que s sea positivo y entonces decrementa s. V es usada para marcar la ocurrencia de un evento y P es usada para aguardar hasta que un evento ocurra.

Declaraciones y operaciones

La declaración consiste de una o más semáforos separados por comas

sem sem1, sem2, . . .

Cada semáforo especifica ya sea un semáforo sencillo o un arreglo de semáforos y opcionalmente los valores iniciales, cada uno de los semáforos tiene la siguiente forma

sem_id dimensión := expr

La dimensión e inicialización son opcionales. El valor de inicialización no debe ser negativo, por omisión es cero.

Las operaciones P y V tiene la siguiente forma

P(sem subíndices)
V(sem subíndices)

ejemplo

Considérese el problema de las secciones críticas; se tienen N procesos que comparten una variable (un contador) para accesar el contador se restringe a un proceso a la vez para asegurar que se actualiza automáticamente

```
resource CS()
  const N:= 20
  var x:= 0
  sem mutex := 1
  process p(i:=1 to N)
  .# sección no crítica
  #sección crítica
  P(mutex)
    x:=x+1
  V(mutex)
  . #sección no crítica
  end
end
```

El semáforo se inicializa a 1 así que solo un proceso a la vez puede modificar a x.

Los procesos en espera se basan en el orden de primero que llega primero que se atiende, de acuerdo a como se ejecuten las operaciones P. Así los procesos esperando son tratados de la siguiente manera: un proceso esperando en un semáforo eventualmente podrá proseguir después de ejecutar una operación P, asumiendo que un número de suficiente de operaciones V son ejecutadas en ése semáforo.

Los arreglos de semáforos se usan a menudo en el manejo de recurso o para controlar familias de procesos. Típicamente un semáforo se asocia con cada recurso o con cada proceso.

Supóngase que N process van a entrar a su sección crítica en un orden circular de acuerdo a su identidad, es decir, primero el process1 después el process2 y así hasta el processN, donde se repite el ciclo.

```
resource CS_ordered()
  const N:=20
  sem mutex[N]:= (1, [N-1] 0)
  process p(i:=1 to N)
    #sección no crítica
    . . .
  #sección crítica
  P(mutex[i])
. . .
  V(mutex[i])
  #sección no crítica
. . .
  end
end
```

El arreglo de semáforos *mutex* tiene un elemento para cada process p. Este actúa como un semáforo binario dividido. A lo más uno de los semáforos en el arreglo es 1, los demás son cero, esto para que solo un process a la vez pueda estar en su sección crítica. El elemento que esté en 1 indica cual process tiene permiso de entrar a su sección crítica. Cuando un process deja su sección crítica, pasa el permiso al siguiente process señalado por mutex[(i mod N) + 1].

Sincronización en barrido

El barrido es una herramienta común de sincronización usada en algoritmos paralelos, algoritmos iteractivos, que requieren que todas las tareas se ejecuten en una iteracción antes de empezar la siguiente.

Una posible estructura para un algoritmo paralelo iteractivo, es emplear diversos process trabajadores y un process coordinador. Los trabajadores solucionan parte de un problema en paralelo. Iteractúan con el coordinador para asegurar la coordinación en barrido.

```
resource barrier()
  const N:=20
  sem done :=0, continue[N] := ([N] 0)
  process worker (i:=1 to N)
    do true ->
      #código donde se hace la tarea
      V(done)
      P(continue[i])
    od
end

process coordinator
  do true ->
    fa w:=1 to N -> P(done) af
    fa w:=1 to N -> V(continue) af
  od
  end
end
```

Cada trabajador ejecuta alguna acción, después ejecuta una V y una P, en ese orden. La señal V indica al coordinador que el trabajador ha finalizado su tarea; la P retarda al trabajador hasta que el coordinador informa que todos los otros trabajadores han completado sus tareas (iteracción). El process coordinador consiste de 2 ciclos for all. El primero espera a que cada trabajador indique que ya acabó. El segundo indica que los trabajadores pueden continuar.

Los semáforos pueden ser declarados en el cuerpo o spec de una global. Dentro de el cuerpo de la global, los semáforos se usan de la misma manera que dentro del cuerpo de un recurso (para sincronizar process en el cuerpo de la global), sin embargo los semáforos declarados en la spec de la global se usan un poco diferente, en éste caso proveen sincronización para process ejecutando dentro o fuera de la global, posiblemente en varios recursos.

```
global coordinator
  const N:=20
  sem continue[N]:= ([N] 0)
  sem done:= 0
  body coordinator
process c # process coordinador
  do true ->
    fa w:=1 to N -> P(done) af
    fa w:=1 to N -> V(continue[w]) af
  od
end
end
```

El recurso principal contiene solo los process trabajadores

```
resource worker()
  import barrier
process worker (i:= 1 to N)
  do true ->
    # código para hacer la tarea
    V(done)
    P(continue[i])
  od
end
end
```

La ventaja de usar una global es que separa los detalles del process coordinador de los de los trabajadores. De hecho los trabajadores podrían ser por sí mismos recursos diferentes.

Paso de mensajes asíncronos

El paso de mensajes en SR se realiza al tener process que mandan mensajes y operaciones que reciben tales mensajes. El emisor del mensaje continua después de mandar el mensaje. El receptor del mensaje espera hasta que haya un mensaje en la cola y entonces lo remueve. Así la sentencia send es asíncrona (no bloquea) y la sentencia receive es síncrona (bloquea).

Un papel alternativo de una operación es definir una cola de mensajes. En ésta caso la operación no tiene un proc correspondiente. En su lugar, invocaciones de la operación (mensajes) son atendidas por la sentencia receive dentro de uno o más process en el alcance de la declaración de la operación. Una sentencia receive elimina una invocación de la cola de mensajes, el process ejecutando espera si no hay invocaciones presentes.

La invocación de un send de una operación atendida por una sentencia receive provoca que la invocación sea agregada a la cola de mensajes. El invocante continua inmediatamente después que la invocación ha sido enviada.

Una sentencia receive usa una operación y una lista de cero o más variables separadas por comas:

receive operación[índices](var1, var2 , . . .)

Los índices se usan para arreglos, se especifica una operación para cada parámetro en la definición de la operación, deben coincidir los tipos correspondientes en los parámetros.

Ejemplo:
Se tienen dos process cada uno manda un flujo ordenado de mensajes a un tercero, el cual imprime la combinación de los dos flujos

```
resource stream_merge()
  const EOS := high(int)
  op stream1 (x: int), stream2(x: int)
process one
. . .
  send stream1(y)
. . .
  send stream1(EOS)
end

process two
. . .
  send stream2(y)
. . .
  send stream2(EOS)
end

process merge
  var v1, v2 int
  receive stream1(v1)
  receive stream2(v2)
  do v1 < EOS or v2 < EOS ->
    if v1 <= v2 -> write(v1)
    receive stream(v1
    [] v2 <= v1 -> write (v2)
    receive stream(v2)
```

```
    fi
  od
  write (EOS)
  end
end
```

Se usan dos operaciones stream1 y stream2. El primer process manda su número, incluyendo la maraca de fin de flujo EOS a stream1, el segundo a strema2. El process merge primero obtiene un número de cada uno. Ejecuta el loop en tanto ninguno de los dos números sea EOS. La sentencia if compara los dos números, imprime el menor y recibe el siguiente número en el flujo para el cual vino el número más pequeño.

Modelos cliente-servidor

Un servidor es un process que maneja en forma repetida requerimientos de process cliente.

Considérese el caso de un cliente y un servidor.

```
resource CS1()
  op request(. . .), result(. . .)
process client
. . .
  send request( . . .)
  receive result(. . .)
. . .
end

process sever
  do true ->
    receive request(. . . )
. . .
    send result(. . . )
  od
  end
end
```

Este código no funcionará para más de un cliente. Una forma de generalizar el código es usar un arreglo de operaciones result, una para cada process cliente.

```
Resource CS2()
  const N:=20
op request (id: int; . . .), result[N](. . .)
  process client(i := 1to N)
. . .
  send request (i, . . .)
. . .
  receive result[i](. . .)
end

process server
  do true ->
    receive request(id, . . .)
. . .
    send result[i](. . .)
  od
  end
end
```

Cada process cliente pasa su identidad como parte del mensaje de solicitud. La identidad se usa por el servidor para mandar de regreso el resultado de una solicitud al cliente que la hizo. Cada process cliente recibe un elemento de la operación result que corresponde a su identidad.

Un inconveniente del resource CS2 es que se requiere conocer el número de clientes. El arreglo result provee un medio sencillo para asociar una operación con cada process cliente. Otra forma de obtener el mismo resultado es declarar una operación local para cada process cliente:

```
resource CS3()
  optype result_type = (. . .. . .. . .. . . )
  op request (result_cap: cap result_type; . . .)
  const N := 20
  process client(i := 1 to N)
    op results : result_type
. . .
    send request(results, . . .)
    # aquí hace algo
    receive results( . . . )
. . .
end

process server
  do true ->
    var results_cap: cap result_type
    receive request( result_cap, . . . )
    #procesa la solicitud
    send result_cap(. . .)
  od
  end
end
```

El tipo operación result_type especifica la parametrización del mensaje de resultado. Cada cliente declara una operación local, result, cuya parametrización está dada por result_type.

El cliente para una capacidad para esa operación como primer parámetro de request. El servidor recibe esa capacidad en la variables local result_cap y la usa para regresar el resultado a la operación a la cual la capacidad apunta.

Una ventaja importante de éste código es que permite a cualquier process cliente interactuar con el servidor. Todo lo que el process cliente necesita es pasar al servidor una operación result. Los clientes pueden estar en diferentes recursos, aún en diferentes máquinas virtuales, en tanto que la operación request es visible al declararla en el spec del recurso server.

Operaciones compartidas.

Las operaciones compartidas son casi una necesidad dado que múltiples instancias de un process pueden servir a la misma operación. Una operación compartida puede ser usada para permite a múltiples servidores atender a misma cola de trabajo. La solicitud de los clientes se atiende al invocar una operación compartida. Los process servidores esperan a los invocaciones de la operación compartida, qué servidor recibe y atiende una invocación particular es transparente para el cliente. Las operaciones compartidas pueden ser declaradas al nivel máximo de los recursos o en una global. Si la operación es declarada en un recurso, los procesos servidores deben estar en la misma instancia del recurso. Si la operación se declara en una global, los process servidores pueden estar en cualquier recurso o global que la importe.

Ejemplo:

Considérese el método de cuadratura adaptativa para encontrar el área bajo la curva. Dada una función continua y los valores l y r con l < r. El problema es calcular el área limitada por f(x). El siguiente recurso describe la solución. Emplea una operación compartida (bag) la cual contiene una bolsa de tareas. Cada tarea representa un subintervalo sobre la cual la integral de f se aproxima.

```
resource main()
  op bag(a, b, fofa, fofb: real)
  op result(area: real)
  var area: real := 0.0
  procedure f(X: real) returns fx:real
...
end
process administrator
  var l, r, part : real
  # inicializa l y r
  send bag (l, r, f(l)m f(r))
  do true ->
    receive result(part)
    area +:= part
  od
end
const N:= 20
process worker(i:= 1 to N)
  var a,b,m, fofa,fofb, fofm : real
  var larea, rarea, tarea, diff: real
  do true ->
    receive bag(a, b, fofa, fofb)
    m:=(a+b)/2
    fofm:= f(m)
    #calcular larea, rarea, tarea
    #usando la regla del trapecio
    diff := tarea - (larea + rarea)
    if . . ./* diferencia suficiente pequeña */
      send result(larea + rarea)
    [] . . ./* diferencia muy grande */
      send bag(a, b, fofa, fofm)
      send bag(m, b, fofm, fofb)
    fi
  od
end
final
  write("área es: ", area)
end
end
```

Inicialmente el administrador pone en la bolsa una tarea correspondiente al problema total. Múltiples process trabajadores toman tareas de la bolsa y las atienden, a menudo generando dos nuevas tareas (correspondientes a subproblemas) las cuales son puestas en la bolsa. Específicamente un trabajador toma una tarea (intervalo[a, b]) de la bolsa, calcula el punto medio m y calcula tres áreas. Estas las de los tres puntos definidos por el trapecio definidos por los puntos a, b, m y el valor de f en esos tres puntos. El trabajador compara el área del trapecio más largo con la suma de las áreas de los trapecios más pequeños. Si éstas son lo suficiente pequeñas la suma de las áreas se toma como una aproximación aceptable del área bajo f y los trabajadores la mandan al administrador usando la operación result. De otra forma el trabajador agrega a la bolsa los dos subproblemas de calcular el área de a m y de m a b

Después de incializar la bolsa de tareas, el administrador repetidamente recibe resultados, los cuales son parte del área total. Este último suma el área, el cálculo termina cuando la cola de mensajes esta vacía y todos los process están bloqueados, es decir todas las tareas han sido procesados por

los trabajadores y todos los resultados han sido recibidos por el administrador. En éste punto se ejecuta el código final. La variable área se declara como una variable del recurso así que es accesible al administrador y al código final.

Un aspecto interesante de éste algoritmo es que permite cualquier número de trabajadores. Si sólo hay uno, el algoritmo es esencialmente iteractivo y secuencial. Si hay más trabajadores, los subproblemas pueden ser resueltos en paralelo. Así el número de trabajadores puede ser adecuado al hardware en el cual el algoritmo ejecute.

Llamadas a procedimientos remotos.

Se involucran dos procesos: el que hace la llamada (el invocante o cliente) y el proceso que atiende la llamada (el servidor). El proceso invocante espera a que el resultado sea regresado de la llamada. Así la llamada a procedimientos remotos es síncrona desde el punto de vista del cliente.
Para iniciar una llamada el process invocante llama a una operación que es atendida por un proc. Una invocación call de un proc remoto resulta en un process creado para atender la invocación. El hecho de que el proc que atiende una llamada esté localizado en una máquina virtual o física diferente es transparente para el llamador.

Mecanismos para llamadas a procedimientos remotos

En todos los casos la semántica de una invocación call a un proc es que un nuevo process se crea para ejecutar el código. Del proc para la invocación. Después de iniciar el call, el process invocante espera hasta que el process invocado regresa. Un nuevo process se crea para atender cada call.

Esta forma de ver las invocaciones call es útil debido a que el proceso invocado puede estar localizado en otro recurso, el cual puede estar localizado en otra máquina física o virtual.

```
resource stack()
  type result=
    enum(OK,OVERFLOW,UNDERFLOW)
  op push (item: int) returns r: result
  op pop ( res item : int) returns r: result
  body stack(size: int)
  var store[1: size]: int
  top : int := 0
proc push (item) returns r
  if top < size -> store [++top]:= item
    r:= OK
  [] top = size -> r:= OVERFLOW
  fi
end

proc pop(item) returns r
  if top > 0 -> item := store[top --]
    r:= OK
  [] top =0 -> r:= OVERFLOW
  fi
end
end stack

resource stack_user()
  import stack
  var x: stack.result
  var s1, s2 : cap stack
  var y: int
  s1 := create stack(10)
  s2 := create stack(20)
...
  s1.push(4)
  s1.push(37)
  s2.push(98)
  if (x:= s1.pop(y)) != OK -> . . . fi
  if (x:= s2.pop(y)) != OK -> . . . fi
..
end
```

Las invocaciones de push y pop de stack_user son invocaciones call a operaciones atendidas como procs en un recurso diferente, las instancias de stack creadas por stack_user se localizarán en la misma máquina virtual, la de stack_user. Sin embargo, las instancias de stack podrían estar en otra máquina virtual en cuyo caso la llamada será remota.

Equivalencia con el par send/receive

Una llamada a un proc remoto puede ser escrita equivalentemente como un send a un proc para crear el process además un receive para regresar resultados

```
op p(val x: int; var y: int; res z: int)
  process q
  var a, b, c: int
...
  call p(a, b, c)
end
proc p(x, y, z)
  z:= x+4
  y-:=10
end
```

Este código puede ser escrito como:

```
op p(x, y : int), r(y, z : int)
  process q
  var a, b, c :int
...
  send p(a, b)
  receive r(b, c)
end
proc p(x, y)
  var z: int
  z:= x+4
  y-:=10
  send r(y, z)
end
```

La operación p ha sido remplazada por una nueva versión de p y una nueva operación resultado, r. La llamada a el proc ha sido remplazada por un par send receive. El send pasa el valor y los parámetros variables a p; el receive regresa la variable y los parámetros de result a p.

Si más de un proceso invoca a p entonces cada uno necesita su propia operación de resultado

```
op p(val x: int; var y: int; res z: int)
  process q(i:=1 to . . .)
  var a, b, c : int
...
  call p(a, b, c)
end
proc p(x, y, z)
  z:= x+4
  y-:= 10
end
```

El código puede ser reescrito:

```
op p(x, y: int; rcap: cap(y, z : int))
  process q(i:=1 to . . .)
  var a, b, c : int
op(y, z : int)
  send p(a, b, r)
  receive r(b, c)
end
proc p(x, y, rcap)
  var x: int
  s:= x+4
  y-:= 10
  send rcap(y, z)
end
```

La operación p se reemplaza por una nueva versión, el tercer argumento es una capacidad para una operación que usada para regresar dos resultados. Cada process invocante declara una operación local r. De nuevo la llamada el proc ha sido remplazada por el par send/receive. Un process emisor pasa la capacidad de su r a p; p manda el resultado a esa operación-

En general, una invocación call provee una interface más clara que el par send/receive. En particular, el paso del resultado al invocante es una parte implícita de una invocación call. Usando el par send/receive, de otra forma, requiere declarar una operación local para la cual el resultado obtiene y manda la capacidad para esa operación. Usando una invocación call también permite invocaciones de operaciones que regresan resultados dentro de las expresiones. Sin embargo, el par send/receive es útil cuando un cliente quiere hacer el trabajo entre el inicio de la solicitud de un servicio y la obtención del resultado que se requirió.

Sentencias return, reply y forward

Return

En algunos ocasiones es útil permitir que un proc sea invocado por un call o un send. En algunos casos el process invocante querrá esperar a que la actualización se complete, en otros no. Tal proc podría ejecutar una sentencia return. Si el proc fue invocado por un send, la sentencia return solo al termina el process que ejecuto el return. Debido a que una invocación send termina inmediatamente después de que los parámetros se mandan al proc, cualquier resultado del proc no es realmente regresado.

Reply

Se usa por un proc para continuar la ejecución después de atender una invocación del proc

reply

Termina la invocación que está siendo atendida por el proc que la contiene. Un process que ejecute una sentencia reply continua ejecutando con la siguiente sentencia al reply, sin embargo los cambios subsecuentes a los parámetros formales o el valor de regreso. Un reply a una invocación send no tiene efecto; un reply para una reply ya ejecutado tampoco tiene efecto.

Puede aparecer en el tope del código de inicialización en un recurso o global. Tal código es ejecutado por un process de inicialización implícitamente creado. Si tal process ejecuta un reply, entonces se realiza la creación del recurso o global y el process creador y de inicialización ejecutan concurrentemente.

```
resource fun()
  op f(x: int) returns y: int
  process p
...
  z:= f(10)
...
end
proc f(x) returns y
  y:= x*8
  reply
  y:= 0
  end
end
```

La sentencia reply en f termina la invocación desde p. A éste punto, el valor de retorno es 80 y se regresa a z, el process p continua la ejecución de la siguiente sentencia. El process que está ejecutando f también continua con la siguiente sentencia al reply. Se modifica el valor de y, entonces termina. Esta última modificación no tiene efecto para el llamador, en particular no cambia el valor de z.

Forward

La ejecución de forward toma la invocación de operación que está siendo atendida, evalúa un nuevo conjunto posible de argumentos e invoca a la operación invocada.

Forward operation (expr, expr, . . .)

Una invocación puede ser adelantada a cualquier operación que tenga la misma nomenclatura, incluyendo la operación que está siendo atendida. Si la invocación que está siendo atendida fue llamada el llamador permanece bloqueado hasta que la nueva invocación sea completada.

Después de ejecutar el forward, process que la ejecuta continua con la siguiente sentencia. Un forward posterior de la misma invocación es tratada como si fuera una invocación send desde el process que la ejecutó. Un reply sobre una invocación forward no tiene efecto

```
resource fun()
  op f(x: int) returns z : int
  op g(y: int) reurns z: int
  process p
  var a:= f(1)
  write(a)
end
proc f(x) returns z
  forward g(2*x)
. . .# continua ejecutando, además cambia z
end
proc g(y) returns z
  z:= y+10
end
end
```

Primero el process p invoca a f. El process ejecutando f duplica su argumento, aplica forward a la invocación a g y entonces continua ejecutando el process puede asignar a z, pero no tiene efecto en el resultado que se regresa a p. Aplicando forward g provoca que un nuevo process se cree; suma 10 a su argumento y regresa su valor a p, el cual está esperando por la invocación de f a que regrese. El efecto final es que la variable a es igual a 12.

Rendezvous

Como la llamada a procedimiento remoto, involucra dos process uno que invoca y otro que atiende la invocación. Sin embargo la invocación es atendida por un process ya existente; no se crea un nuevo process como resultado de la invocación, rendezvous es síncrono desde la perspectiva del invocante y los dos process pueden estar en diferentes máquinas físicas o virtuales.

Se realiza a través del uso de operaciones. La operación que es atendida por un process existente ejecuta la sentencia input, la cual permite que un process espere a que una o diversas operaciones sean invocadas. También permite que un process base su decisión de cual invocación atender de acuerdo a los valores de los parámetros de invocación.

La sentencia input

Una sentencia input contiene un o más comandos de operación separados por []

in op_comando[] opcomando[] . . .ni

Cada comando de operación especifica una operación a atender, una cláusula opcional de sincronización, una cláusula opcional de orden y un bloque de código. Un comando de operación que atiende a una operación sin valor de regreso tiene la forma:

```
operation( formal_id_list )
        st sync_expr by sched_exp -> block
```

Un comando de operación que atiende una operación con valor de regreso tiene la forma:

operation (formal_id_list) returns result_ide

st synch_exp by sched_exp ->block

Los identificadores para los parámetros y el valor de retorno son nombres nuevos. La palabra clave st (such that) introduce la expresión de sincronización, especifica cual invocación de la operación es aceptable, la palabra clave by introduce la expresión de orden, indica el orden en que las invocaciones son atendidas.

En general, una sentencia input puede atender cualquier operación declarada en el alcance que incluya la sentencia. La misma operación puede aun aparecer en más de un comando de operación declarada en un global puede atender solo operaciones declaradas en esa global.

La operación, en el comando de operación puede ser elemento de un arreglo de operaciones. Un proceso ejecutando una sentencia input es en general detenido hasta que alguna invocación sea seleccionable. Una invocación es seleccionable si la expresión de sincronización es evaluada en forma booleana para la correspondiente operación es cierta. En general la invocación seleccionable más atrasada es la que se atiende. Sin embargo si el comando de operación correspondiente contiene una expresión de orden, la invocación que es atendida es la más antigua seleccionable y que también minimice la expresión de orden. Ambas sincronización y orden pueden referenciar parámetros de invocación, al permitir que la selección sea basada en sus valores. Si o hay invocaciones pendientes para una sentencia input, el process que ejecuta la sentencia espera hasta que se recibe una invocación seleccionable.

Una invocación es atendida el ejecutar el bloque correspondiente. La sentencia input termina cuando ese block termina, si la invocación fue llamada, la sentencia call correspondiente también termina.

Los comandos de operación en una sentencia input puede ser seguidos por comandos else:

[] else -> block

Este bloque de código se ejecuta si la invocación es no seleccionable. Así un process nunca esperará cuando se tenga un comando else.

Sentencias input sencillas

Como ejemplo dos process p y q interactúan vía rendezvous a través del process f. El process p llama a f(y) y entonces espera hasta que q recibe a y e incrementa su variable local z; esto es p espera hasta que q alcance el fin del bloque de comando asociado con f. Si el process q llega a su sentencia input y no encuentra invocaciones pendientes de f, espera hasta que q invoque a f.

```
resource main()
  op f(x: int)
  process p
    var y: int
...
    call f(y)
...
  end
process q
  var z: int
...
  in f(x) -> z+:= x ni
...
  end
end
```

Ejemplo 2

```
resource main()
  op f(x: int), g(u: real) returns r: real
  process p1
    var y: int
...
    call f(y)
...
end
process p2
  var w: real
  w:= g(38)
...
end
process q
  var z: int
...
  in f(x) -> z+:= x
  [] g(u) returns v -> v:= u*u - 93
  ni
...
  end
end
```

Tres process interactúan a través de dos rendesvouz: p1 y q por medio de la operación f; p2 y q interactúan a través de la operación g, la cual tiene un valor de retorno. La sentencia input permite al process q atender ya sea una invocación de f o de g. Cuando q alcanza su sentencia input, ésta se encuentra en uno de tres estados

- Sólo una invocación de f o g está pendiente (q atenderá la operación que tenga invocaciones pendientes)
- Invocación es de ambas f y g están pendientes (q atenderá ña invocación que llegue primero.
- No hay invocaciones pendientes (q espera hasta que f o g sea invocada)

Después de invocar f el process p1 espera hasta que su invocación es atendida por el process q, es decir, hasta que q alcanza el fin del bloque en el correspondiente comando de operación. De manera similar, después de invocar g, el process p2 espera hasta que su invocación es atendida por el process q. Como está programado solo un rendezvous realmente ocurrirá; si la sentencia input fuera introducida en un loop y ejecutada dos veces, ambas podrían ocurrir pero un orden impredecible.

El siguiente ejemplo muestra como se pueden anidar sentencias input, aun cuando sirvan a la misma operación, notar que los identificadores formales usados en una sentencia input no necesitan coincidir con los identificadores de los parámetros usados en la correspondiente declaración de operación.

```
resource main()
  op swap( var x: int)
  process p1
    var y: int
    call swap(y)
...
  end
  process p2
    var z: int
    call swap(z)
...
  end
  process q
    in swap(x1) -> in swap(x2) -> x1:=:x2 ni ni
...
  end
end
```

Los process p1 y p2 invocan a swap para intercambiar valores. El process q usa un input anidado para atender una invocación de swap dentro de otra. Así q atiende una de las llamadas swap con el input exterior y la otra con la sentencia input anidada. En el bloque más interno q tiene acceso a los parámetros de ambas invocaciones de swap aunque ellos tiene diferentes nombres locales.

Expresiones de sincronización

Una expresión de sincronización evaluada en booleano puede ser usada para controlar cual invocación y sentencia input será la siguiente en atenderse

```
in a(x) st c > 0 -> . . . ni
in a(x) st x = 3 -> . . . ni
in a(x) st x = 3 -> . . . ni
[] b(y, z) st y = f(z) -> . . .
ni
```

La primera sentencia input atiende una invocación de a solo cuando el valor de la variable c es positivo. La segunda atiende solo invocaciones de a cuyo parámetro x sea igual a 3. La tercera atiende la misma invocación que la segunda así como invocaciones de b cuyo parámetro satisfaga la condición y = f(z)

Es importante enfatizar que las expresiones de sincronización pueden referenciar parámetros de invocación. Esta habilidad genera soluciones robustas a muchos problemas de sincronización.
El siguiente ejemplo presenta una solución al problema de buffer acotado. El recurso bounded_buffer provee dos operaciones: deposit y fetch. Un productor llama a deposit para insertar un ítem en el buffer, un consumidor llama a fetch para obtener un ítem del buffer. Una sentencia input sincroniza como las invocaciones de fetch y deposit son atendidas, para asegurar que los mensajes son extraídos en el orden en el que fueron depositados no son extraídos hasta que se depositen y no son sobrescritos

```
resource bounded_buffer
  op deposit(item: int)
  op fetch () returns item: int
body bounded_buffer(size: int)
  var buf[0: size -1] : int
  var count := 0, front := 0 rear:= 0
  process worker
    do true ->
    in deposit(item) st count < size ->
      buf[rear] := item
      rear := (rear +1) % size
      count ++
    [] fetch() returns items st count > 0 ->
       item := buf[front]
       front := (front + 1) % size
       count - -
    ni
  od
  end
end
```

El process worker gira en torno a una sentencia input sencilla, la cual atiende a deposit y fetch. La expresión de sincronización en la sentencia input asegura que el buffer no caiga en overflow o underflow. Por ejemplo en productor es detenido si el buffer está lleno y un consumidor es detenido si el buffer está vacío.

En el problema de lectores/escritores hay dos clases de process que quieren accesar el recurso. Los process lectores solo examinan el recurso, aunque pueden ejecutar concurrentemente con otros. Los process escritores actualizan el recurso para mantenerlo consistente, ellos deben tener acceso exclusivo a él. Supóngase que el process lector llama a la operación start_read antes de leer y llama (o manda) una operación end_read cuando acabó. De manera similar el process escritores llama a start-write antes y llama (o manda) una operación end_write después de escribir:

```
process rw_allocator
  var nr:= 0, nw := 0
  do true ->
    in start_read() st nw= -> nr++
      [] end_read() -> nr--
      [] start_write() st nr=0 and nw=0 -> nw++
      [] end_write() -> nw--
    ni
  od
end
```

Las variables nw y nr cuentan el número de lectores y escritores activos, respectivamente. El lector puede empezar a leer cuando no hay escritores activos; un process escritor puede empezar a escribir cuando no haya lectores activos o escritores.

Una función predefinida, denotada por el operador prefijo ? regresa el número de invocaciones pendientes para esa operación, ?f regresa el número de invocaciones pendientes de la operación f, puede ser usada solo dentro del alcance de f. Esta función puede ser usada para preferenciar la atención de una operación sobre otra:

```
in f(. . .) -> . . .
[] g(. . .) st ?f = 0
ni
```

La expresión de sincronización en la segunda parte de la sentencia input dice que una invocación de g debe ser atendida solo si no hay invocaciones pendientes de f.

Expresiones de orden

Una expresión de orden puede ser usada para controlar cual invocación de la sentencia input atenderá. Se le da preferencia a la invocación seleccionable que minimiza la expresión de orden, si hay más de una invocación se selecciona la primera que se haya hecho. El tipo de expresión de orden puede ser cualquier tipo ordenado. Las expresiones de orden facilitan el resolver problemas, pero incurren en un costo de implantación ya que todas la invocaciones pendientes tiene que ser examinadas.

```
in a(x) by x -> . . .ni
in a(x) st x mod 2 = 0 by -x -> . . . ni
in a(x) st x mod 2 = 0 by -x -> . . .
[] b(y, z) by y+z -> . . .
ni
```

El primer in atiende las invocaciones de a, dando preferencia a aquellas con valores de x pequeños. La segunda atiende invocaciones de a cuyo parámetro x es impar, dando preferencia a aquellas con valores de x grandes. La tercera atiende las mismas invocaciones de la segunda en el mismo orden o atiende invocaciones de b dando preferencia a aquellas con valores de z+y pequeños.

Exactamente cuál invocación se atenderá en la tercera sentencia input depende de qué invocaciones estén pendientes y cuándo llegaron. El caso más interesante ocurre cuando invocaciones de a para las cuales x es impar y hay invocaciones de b pendientes. De esas invocaciones la que llegue primero determina cuál es atendida. Si una de esas operaciones es a, la invocación con el valor de x más grande es atendida (aún si hubiera invocaciones de b anteriores). De otra forma la invocación de b con el valor de y+z más pequeño es atendida.

Input condicional

El bloque de código asociado a un else se ejecuta si ninguna de las operaciones guardia de la sentencia input es verdadera. El uso del else soporte el input condicional.

```
in -> x:0 1 [] else -> x:0 2 ni
```

Si una invocación de a se presenta, el proceso ejecuta la primera asignación, sino ejecuta la segunda. Un input con un else nunca provoca que el process se bloquee.

```
do true ->
  #hace algo
  in done() -> exit [] else -> skip ni
```

```
od
```

Un process repetidamente hace algún trabajo y entonces checa por una invocación de la operación done indicando que debería terminar. Si no hay tal invocación el process continua con la siguiente iteración.

```
do true ->
  in a(x) st x=t -> . . . [] else -> exit ni
od
```

El efecto del ciclo do es atender todas las invocaciones pendientes de a cuyo parámetro sea igual a t. En cada iteración del ciclo la sentencia input atiende una de esas invocaciones si hay alguna o sale del ciclo.

Arreglos de operaciones

La sentencia input puede ser usada también para atender arreglos de operaciones.

```
op f[N](x: int)
  process q(i:= 1 to N)
. . .
    in f[i](x) -> . . . ni
. . .
  end
```

Cada process atiende un elemento del arreglo t.

Para atender cualquiera de un grupo de elementos:

```
in (i:= 1 to 4) signal[i]() -> skip ni
```

Máquinas virtuales

Cada máquina virtual defina un espacio de direcciones en una máquina física, cuando se crea una máquina virtual puede ser colocada en una máquina física específica.

El uso de una máquina virtual se refleja cómo se crean las instancias de recursos y globales. Los process, variables y operaciones de una instancia del recurso existen solo dentro de la máquina virtual, una instancia de una global se crea en cada una de las máquinas virtuales que la necesite. La comunicación entre máquinas virtuales es transparente.

Inicio y ejecución de un programa

La ejecución de una programa empieza con la creación implícita de una máquina virtual, en la cual se crea una instancia del recurso principal. Esta máquina virtual principal se ejecuta en la máquina física en la cual se inicio la ejecución del programa.

Creando una máquina virtual.

Una máquina virtual se crea al crear una instancia del pseudorecurso vm, el cual es un recurso especial y predefinido. En éste caso la sentencia create regresa una capacidad del tipo cap vm

```
var c: cap vm
c:= create vm()
```

Por omisión una máquina virtual nueva se pone en la misma máquina física de su creador. Una nueva máquina virtual puede ser puesta en una máquina física específica:

```
create vm() on expr
```

Donde expr específica una máquina física como una cadena o un entero. Cuando es un entero representa el mapeo entre el valor y una máquina específica

```
var c1, c2, c3 : cap vm
c1:= create vm()
c2:= create vm() on "uxmcc2"
c3:= create vm() on "tonatiuh"
```

Creando recursos y globales

Por omisión la instancia de un recurso se crea en la misma máquina virtual que su creador. La siguiente expresión crea una instancia de un recurso en una máquina virtual existente

```
create res_name(argumentos) on expr
```

Expr es una capacidad para la máquina virtual en la cual la instancia del recurso se creará

```
create r(34) on c1
create r(22) on c2
create r(70)
```

La globales son específicas a la máquina virtual. Cada una contiene su propia instancia de global si la necesita. En particular una instancia de una global se crea implícitamente en una máquina virtual la primera vez que un recurso o global

la importa. Las variables y operaciones declaradas en la spec de una global son locales a la máquina virtual.

Destruyendo máquinas virtuales

Se destruyen usando:

destroy expr

expr es un tipo cap vm

Cuando se destruye una máquina virtual se previene la creación de un nuevo recurso en ella. Entonces cada recurso existente en la máquina virtual se destruye, se ejecuta el código final y el espacio de direcciones que ocupa se libera.

```
global glob
  var x:= 0
  sem mutex := 1
body glob
  final
  write (x)
end
end
```

Si la global fuera usada en la máquina virtual principal su código final se ejecuta cuando el progrma termina. Si se usa en otra su código final se ejecuta cuando esa máquina virtual se destruye.

```
resource foo( N, n: int; c: cap())
  import glob
process p(i:=1 to N)
  P(mutex)
  x+:= n
  V(mutex)
  send c()
  end
end
```

La actualización de x se protege usando mutex. Cada process manda un mensaje a la operación apuntada por la capacidad c, que es un parámetro del recurso.

```
resource main1()
  import foo
  const M:= 5
  op done()
  var foo1, foo2 : cap foo
  foo1 := create foo (N, 1, done)
  foo2 := create foo (N, 2, done)
  fa i:=1 to 2*N -> receive done() af
  destroy foo1
  destroy foo2
end
```

Se crean dos instancias de foo las cuales comparten los mensajes done de cada process p en cada instancia y finalmente destruye las dos instancias de foo y el programa termina

Este programa ejecuta en una sola máquina física por lo tanto solo se crea una instancia de la global.

```
resource main2()
  import foo
  const N:= 5
  op done(); var vmcap: cap vm
  var foo1, foo2 : cap foo
  foo1 := create foo(N, 1, done)
  vmcap := create vm()
  foo2 := create foo(N, 2, done) on vmcap
  fa i:=1 to 2*N -> receive done() af
  destroy foo1
  destroy foo2
 destroy vmcap
end
```

Se crea una segunda máquina virtual en la cual se pone una instancia de foo.

Debido a que se ejecuta en dos máquinas virtuales una instancia de global se crea en cada una. Se crean instancias separadas de x y de mutex.

El programa imprime primero el número 10 (cuando se destruye la segunda máquina virtual) y después el número 5 cuando el programa termina. Este programa se ejecuta en una sola máquina física, si se desea tener otra sería necesario cambiar:

vmcap := create vm() on "uxmcc2"

Entrada/Salida argumentos en línea

La máquina virtual inicialmente hereda los flujos de entrada estándar (stdin) salida estándar (stdout) y el error estándar (stderr) desde el comando de inicia el programa. Una máquina virtual creada por el programa hereda stdout y stderr de la inicial pero el stdin se conecta a /dev/null

La entrada y salida es específica a la máquina virtual. En particular, las capacidades para archivos que han sido abiertos en una no serán válidas en otra.

Las funciones que accesan argumentos en línea (numargs y getargs) son válidas solo para la máquina virtual principal. Así los argumentos en línea no pueden ser accesados directamente desde otra. Si es necesario los datos deben ser pasados desde la principal.

BIBLIOGRAFIA

BIBLIOGRAFIA

[i] Peter Wegner "Guest Editorr's Introduction to Special Issue Computating Surveys." *ACM Computing Surveys*, 21, 3 (Sept. 1989), 253-257.

[ii] Henri E. Bal, Jennifer G. Steiner, Andrew S. Tanenbaum Programming languages for distributed computing systems; ACM Computing Surveys, Vol 21, No 3, september 1989.

[iii] How to write parallel programs: A guide to perplexed, Nicholas Carriero, David Gelernter; ACM Computing Surveys, Vol 21, No 3, September 1989

[iv] Andrews, G. R. and Schneider, B. F. "Cancepts and Notations for Concurrent Programming." *Computing Surveys*, 15, 1, (March. 1983) 2-43.

[v] Dijkstra E. W. "The structure of the "THE" multiprogramming system." *Commun*, ACM 11, 5 (May 1968), 341-346.

[vi] Jones, A. K., And Schawrz , P. "Experience using multiprocessor systems - A status report." ACM Computing Surveys, 12, 2 (june 1980), 121-165.

[vii] Floyd, R. W."Assining meaning to programs." *In Proc. Am. Math. Soc. Symp. Applied Mathematics*, vol. 19, pp. 19-31, 1967

[viii] Hoare. C. R. "An axiomatic basis for computer programming." *Commun. ACM.* 12, 10 (Oct. 1969), 576 580, 583.

[ix] Akkoyunnlu, E, A,, A. Bernstein , A. J., Schneider, F. B., and Silberschatz, A. "Conditions for the equivalence of synchronous and asynchronus systems." *IEEE Trans. Softw. Eng.* SE-4, 6 (Nov 1978), 507-516.

[x] Dennis, J. B. And Van Horn, E. C. "Programming semantics for multiprogrammed computations." *Commun, ACM*), 3 (March 1966), 143-155.

[xi] Peterson, G. L., "Myths about the mutual exclusion problem." *Inform. Process. Lett.* 12,3 (june 1981), 115-116.

[xii] Hoare, C. A. R. "Towards a theory of parallel programming." In C. A. R. Hoare and R. H. Perrot (Eds,), *Operating Systems Techniques*. Academic Press, New York, 1972, pp. 61-71.

[xiii] Brinch Hansen, P. "Structured multiprogramming." *Commun. ACM* 15,7 (July 1972), 574-578.

[xiv] Dijkstra, E. W. "Cooperating sequential processes." In F. Genuys (Ed.), *Programming Languages*, Academic Press, new York, 1968.

Brinch Hansen P. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, N. J., 1973.

Hoare, C. A. R. "Monitors: An operating system structuring concept." *Commun. ACM* 17, 10 (Oct. 1974), 549-557.

[xv] Parnas, D. L. " On the criteria to be used in decomposing systems in to modules." *Commun. ACM* 15, 12 (Dec. 1972), 1053-1058.

[xvi] Dijkstra, E. W. "Guarded commands, nondeterminancy, and formal derivation of programs." *Commun. ACM* 18, 8 (Aug. 1975), 453-457.

[xvii] Andrews, G. R. "Synchronizing resources." *ACM Trans. Prog. Lang. Syst.* 3,4 (Oct 1981), 405-430.

[xviii] Francois L'Hote et al "Robot Components and Systems." *Prentice Hall, Inc Englewood Cliffs*, 1983, pp 9.

[xix] Eduardo Ríos *"Diseño ey Construcción de Circuitos Acopladores"* Laboratorio de Sistemas Digitales y Microprocesadores del Colegio de Electrónica Benemerita Universidad Autónoma de Puebla (Jul 1993)

[xx] Newton C. Braga *"Control de Motores en Robots Paso a Paso"* Saber Electrónica, Año 3 No 13, pp 5-10, 1993

[xxi] Jack Tackett and David Gunter "Special Edition Using Linux", Second Edition. *QUE*, 1996, pp. 13, 20.

[xxii] Ronald A olsson Et. Al. "SR A Language for Parallel and Distributed Programming". *Department of computer science*, The University of Arizona, Tucson Arizona, 1992

[xxiii] Dijkstra E. W. "Cooperating sequential processes." In *Programming Languages*. Academic Press, NY, 43-112, 1968.

[xxiv] Horacio D. Vallejo *"Control Remoto Para Motores Paso a Paso"* Saber Electrónica Año 6 No 9 pp 6-13 Sep 1995