Conceptos básicos

Un programa concurrente especifica dos o más procesos que cooperan para realizar una tarea. Cada proceso es un programa secuencial que ejecuta una secuencia de sentencias. Los procesos cooperan por comunicación: se comunican usando variables compartidas o pasaje de mensajes. Cuando se usan variables compartidas, un proceso escribe en una variable que es leída por otra. Cuando se usa pasaje de mensajes, un proceso envía un mensaje que es recibido por el otro.

Los programas concurrentes son inherentemente más complejos que los secuenciales. En muchos aspectos, son a los programas secuenciales como el ajedrez a las damas: ambos son interesantes, pero el primero es intelectualmente más intrigante. El libro explora el “juego” de la programación concurrente, observando sus reglas, piezas de juego y estrategias. Las reglas son herramientas formales que nos ayudan a entender y desarrollar programas correctos; las piezas son los mecanismos de los lenguajes para describir computaciones concurrentes; y las estrategias son conjuntos de paradigmas de programación útiles.

Las herramientas formales están basadas en razonamiento asercional. El estado de un programa está caracterizado por un predicado llamado aserción, y los efectos de ejecutar sentencias están caracterizados por transformadores de predicado. El razonamiento asercional provee la base para un método sistemático para derivar programas que satisfacen propiedades especificadas.

Se han propuesto muchos mecanismos diferentes para especificar la ejecución concurrente, la comunicación y la sincronización. Se describirán los más importantes (incluyendo semáforos, monitores y pasaje de mensajes) y también nociones de los lenguajes de programación que emplean diferentes combinaciones. Para cada mecanismo de sincronización se muestra cómo desarrollar programas correctos y eficientes, se presentan numerosos ejemplos, y se muestra cómo implementar el mecanismo.

Finalmente se describen paradigmas identificados para programación concurrente. Estos incluyen estrategias de solución y técnicas de programación que son aplicables a una gran variedad de problemas. La mayoría de los programas concurrentes resultan de combinar un pequeño número de estructuras y pueden verse como instancias de unos pocos problemas standard. Aprendiendo estas estructuras y técnicas de solución es fácil resolver problemas adicionales.

Hardware y aplicaciones

La historia de la programación concurrente siguió las mismas etapas que otras áreas experimentales de la ciencia de la computación. Así como el hardware fue cambiando en respuesta a los cambios tecnológicos, las aproximaciones ad-hoc iniciales evolucionaron hacia técnicas de programación generales.

Los sistemas operativos fueron los primeros ejemplos de programas concurrentes y permanecen entre los más interesantes. Con el advenimiento de los controladores de dispositivos independientes en los ‘ 60, se volvió natural organizar un SO como un programa concurrente, con procesos que manejan dispositivos y la ejecución de tareas de usuario. Los procesos en un sistema monoprocesador son implementados por *multiprogramación*, ejecutando un proceso por vez en el tiempo y usando interleaving.

La tecnología evolucionó para producir una variedad de sistemas *multiprocesador*. En un multiprocesador de *memoria compartida*, múltiples procesadores comparten una memoria común; el tiempo de acceso a memoria es uniforme (UMA) o no uniforme (NUMA). En un *multicomputador*, varios procesadores llamados nodos se conectan por hardware de alta velocidad. En un sistema en *red*, varios nodos mono o multiprocesador comparten una red de comunicaciones (por ej, Ethernet). También existen combinaciones híbridas como redes de workstations multiprocesador. Los SO para multiprocesadores son programas concurrentes en los cuales al menos algunos procesos pueden ejecutar en paralelo. Los procesadores varían desde microcomputadoras a supercomputadoras.

Hay muchos ejemplos de programas concurrentes además de los SO. Se dan cuando la implementación de una aplicación involucra paralelismo real o aparente. Por ej, se usan programas concurrentes para implementar:

- sistemas de ventanas en computadoras personales o workstations
- procesamiento de transacciones en sistemas de BD multiusuario
- file servers en una red

- computaciones científicas que manipulan grandes arreglos de datos

La última clase de programa concurrente suele llamarse *programa paralelo* dado que típicamente es ejecutado en un multiprocesador. Un *programa distribuido* es un programa concurrente (o paralelo) en el cual los procesos se comunican por pasaje de mensajes.

Sincronización

Como se mencionó, los procesos en un programa concurrente se comunican usando variables compartidas o mensajes. La comunicación provoca la necesidad de sincronización. En los programas concurrentes se dan dos formas de sincronización: *exclusión mutua* y *sincronización por condición*. La primera consiste en asegurar que las *secciones críticas* de sentencias que acceden a objetos compartidos no se ejecutan al mismo tiempo. La sincronización por condición asegura que un proceso se demora si es necesario hasta que sea verdadera una condición dada. Por ej, la comunicación entre un proceso productor y uno consumidor es implementado con frecuencia usando un buffer compartido. El que envía escribe en un buffer; el que recibe lee del buffer. Se usa exclusión mutua para asegurar que ambos no accedan al buffer al mismo tiempo; y se usa sincronización por condición para asegurar que un mensaje no sea recibido antes de que haya sido enviado y que un mensaje no es sobrescrito antes de ser recibido.

El *estado* de un programa concurrente en cualquier instante de tiempo consta de los valores de las variables de programa. Estas incluyen variables explícitas declaradas por el programador y variables implícitas (por ej, el contador de programa para cada proceso) que contiene información de estado oculta. Un programa concurrente comienza la ejecución en algún estado inicial. Cada proceso en el programa se ejecuta a una velocidad desconocida. A medida que se ejecuta, un proceso transforma el estado ejecutando sentencias. Cada sentencia consiste en una secuencia de una o más *acciones atómicas* que hacen transformaciones de estado indivisibles. Ejemplos de acciones atómicas son instrucciones de máquina ininterrumpibles que cargan y almacenan valores de registros.

La ejecución de un programa concurrente genera una secuencia de acciones atómicas que es algún interleaving de las secuencias de acciones atómicas de cada proceso componente. El trace de una ejecución particular de un programa concurrente puede verse como una *historia*:

$$S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} .... \xrightarrow{a_i} S_i \xrightarrow{a_{i+1}} .....$$

Cada ejecución de un programa concurrente produce una historia. Aún para los programas más triviales el posible número de historias es enorme. Cada proceso tomará distintas acciones en respuesta a diferentes estados iniciales.

Dada esta visión, el rol de la sincronización es restringir las posibles historias de un programa concurrente a aquellas que son deseables. La exclusión mutua concierne a combinar las acciones atómicas fine-grained que son implementadas directamente por hardware en secciones críticas que deben ser atómicas para que su ejecución no sea interleaved con otras secciones críticas que referencian las mismas variables. La sincronización por condición concierne a la demora de un proceso hasta que el estado conduzca a una ejecución posterior. Ambas formas de sincronización causan que los procesos se demoren y por lo tanto restringen el conjunto de acciones atómicas que son elegibles para su ejecución.

Propiedades de programa

Una propiedad de un programa es un atributo que es verdadero en cualquier posible historia del programa, y por lo tanto de todas las ejecuciones del programa. Cada propiedad puede ser formulada en términos de dos clases especiales de propiedades: seguridad (safety) y vida (liveness). Una *propiedad de seguridad* asegura que el programa nunca entra en un estado malo (es decir uno en el que algunas variables tienen valores indeseables). Una *propiedad de vida* asegura que el programa eventualmente entra en un estado bueno (es decir, uno en el cual todas las variables tiene valores deseables).

La *corrección parcial* es un ejemplo de una propiedad de seguridad. Asegura que si un programa termina, el estado final es correcto (es decir, se computó el resultado correcto). Si un programa falla al terminar, puede nunca producir la respuesta correcta, pero no hay historia en la cual el programa terminó sin producir la respuesta correcta. *Terminación* es un ejemplo de propiedad de vida. Asegura que un programa eventualmente terminará (es decir, cada historia del programa es finita). *Corrección total* es una propiedad que combina la corrección parcial y la terminación. Asegura que un programa siempre termina con una respuesta correcta.

La *exclusión mutua* es otro ejemplo de propiedad de seguridad. Asegura que a lo sumo un proceso a la vez está ejecutando su sección crítica. El estado malo en este caso sería uno en el cual las acciones en las regiones críticas en distintos procesos fueran ambas elegibles para su ejecución. La *ausencia de deadlock* es otro ejemplo de propiedad de seguridad. El estado malo es uno en el que todos los procesos están bloqueados, es decir, no hay acciones elegibles. Finalmente, un *entry eventual* a una sección crítica es otro ejemplo de propiedad de vida. El estado bueno para cada proceso es uno en el cual su sección crítica es elegible.

Dado un programa y una propiedad deseada, cómo podemos demostrar que el programa satisface la propiedad? Una aproximación es *testear* o *debuggear*, lo cual puede ser caracterizado como “tome el programa y vea que sucede”. Esto corresponde a enumerar algunas de las posibles historias de un programa y verificar que son aceptables. El defecto del testeo es que cada test considera solo una historia de ejecución específica; un test no puede demostrar la ausencia de historias malas.

Una segunda aproximación es usar *razonamiento operacional*, el cual puede ser caracterizado como “análisis exhaustivo de casos”. Todas las posibles historias de un programa son enumeradas considerando todas las maneras en que las operaciones de cada proceso podrían ser interleaved. Desafortunadamente, el número de historias en un programa concurrente es generalmente enorme (por eso es “exhaustivo”). Para un programa concurrente con n procesos y cada uno con m acciones atómicas, el número de historias diferentes del programa es $(n*m)! / (m!)^2$. Para $n=3$ y $m=2$, hay 90 historias diferentes.

Una tercera aproximación es emplear *razonamiento asercional*, que puede caracterizarse como “análisis abstracto”. En esta aproximación, se usan *aserciones* (fórmulas de predicados lógicos) para caracterizar conjuntos de estados (por ej, todos los estados en que $x > 0$). Las acciones se ven como *transformadores de predicado* que cambian el estado de uno que satisface un predicado a uno que satisface otro. La virtud de este método es que lleva a una representación compacta de estados y transformaciones. Además lleva a una manera de desarrollar y analizar programas en la cual el trabajo involucrado es directamente proporcional al número de acciones atómicas en el programa.

Se empleará la aproximación asercional como herramienta para construir y entender soluciones a una variedad de problemas no triviales. Sin embargo, también se usarán las otras aproximaciones. se usarán razonamiento operacional para guiar el desarrollo de varios algoritmos. Y muchos de los programas en el texto fueron testeados ya que ayuda a incrementar la confianza en la corrección de un programa. Los programas concurrentes son muy difíciles de testear y debuggear ya que es difícil detener todos los procesos a la vez para examinar su estado, y cada ejecución del mismo programa puede producir una historia diferente porque las acciones podrían ser interleaved en orden diferente.

Programación Secuencial

Los programas concurrentes extienden los programas secuenciales con mecanismos para especificar concurrencia, comunicación y sincronización. El objetivo es entender cómo construir programas concurrentes correctos. Aquí se mostrará como construir programas secuenciales correctos.

La primera sección presenta la notación que se usará para programación secuencial. La notación es similar a Pascal, pero las construcciones de control, especialmente para iteración, son más poderosas. Luego se revén conceptos de lógica formal, proposiciones y predicados. Luego se presenta un sistema lógico formal específico que contiene axiomas y reglas de inferencia para probar propiedades de programas secuenciales. Finalmente, se presenta un método (basado en lo que llamamos precondiciones débiles) para derivar un programa y su prueba de corrección total partiendo solo de una especificación del resultado deseado.

NOTACION DEL LENGUAJE

Un programa secuencial contiene *declaraciones*, *sentencias*, y *procedimientos*. Las declaraciones definen tipos, variables y constantes. Las sentencias se usan para asignar valores a variables y controlar el flujo de ejecución dentro del programa. Los procedimientos definen subrutinas y funciones parametrizadas. Los elementos básicos de un programa son identificadores, palabras claves, literales y operadores.

Declaraciones

Los *tipos básicos* son bool, int, real, char, string y enumeración.

Las variables son introducidas por declaraciones **var**.

var id_1 : $tipo_1$:= $valor_1$, ..., id_n : $tipo_n$:= $valor_n$

El identificador id_i es el nombre de una variable con tipo de datos $tipo_i$ y un valor inicial opcional $valor_i$.

Una constante es una clase especial de variable; se le asigna valor una sola vez al declararla. La forma de una declaración constante es la misma que para las variables, excepto que var es reemplazado por **const**.

Un arreglo se declara agregando una especificación de rango a una declaración de variable.

Un tipo registro define un conjunto de valores de datos de tipos potencialmente diferentes

Sentencias

La *sentencia skip*, **skip**, es la sentencia "vacía". Termina inmediatamente y no tiene efecto sobre ninguna variable de programa. Se usa en las sentencias guardadas y las sentencias **await** cuando no hay que tomar ninguna acción cuando una condición booleana se vuelve verdadera.

La *sentencia de asignación*, x := e, evalúa la expresión e y asigna su resultado a x. Los tipos de x y e deben ser los mismos.

La *sentencia de swap* es una clase especial de asignación que intercambia los valores de dos variables. Si v1 y v2 son variables del mismo tipo, entonces

v1 :=: v2

intercambia sus valores.

Una *sentencia compuesta* consta de una secuencia de sentencias, ejecutadas en orden secuencial. Por ej:

x := x + 1; y := y - 1

Las sentencias de alternativa (**if**) e iterativa (**do**) contienen una o más *sentencias guardadas*, cada una de la forma:

$B \rightarrow S$

Aquí, B es una expresión booleana llamada guarda y S es una sentencia simple o compuesta. La expresión booleana B "guarda" a S en el sentido de que S no se ejecuta a menos que B sea verdadera.

Una *sentencia alternativa* contiene una o más sentencias guardadas:

if $B_1 \rightarrow S_1$
□ ...

□ B_n → S_n
fi

Las guardas son evaluadas en algún orden arbitrario. Si la guarda B_i es verdadera, entonces se ejecuta la sentencia S_i. Por ejemplo:

if x ≥ y → m := x
□ y ≥ x → m := y
fi

Si ambas guardas son verdaderas (en el ejemplo, x=y), la elección de cuál se ejecuta es *no determinística*. Si ninguna guarda es verdadera, la ejecución del **if** no tiene efecto. Por ejemplo:

if x < 0 → x := -x **fi**

setea x a su valor absoluto. Si x es no negativo no se ejecuta ninguna sentencia; en otro caso, el signo de x se invierte. El programador puede hacer esto explícito usando **skip** y escribiendo:

if x < 0 → x := -x □ x ≥ 0 → **skip fi**

La *sentencia iterativa* es similar a la alternativa, excepto que las sentencias guardadas son evaluadas y ejecutadas repetidamente hasta que todas las guardas sean falsas. La forma del **do** es:

do B_1 → S_1
□ ...
□ B_n → S_n
od

Como en el **if**, las guardas se evaluadas en algún orden arbitrario. Si al menos una guarda es verdadera, se ejecuta la correspondiente sentencia, y se repite el proceso de evaluación. Como el **if**, el **do** es *no determinístico* si más de una guarda es verdadera. La ejecución termina cuando no hay más guardas verdaderas.

La *sentencia for-all* es una forma especial y compacta de la sentencia iterativa que es útil para iterar a través de los elementos de un arreglo. Su estructura es:

fa cuantificadores → sentencias **af**

El cuerpo del **fa** es una o más sentencias. Cada cuantificador especifica un rango de valores para una variable de iteración:

variable := expr_inicial **to** expr_final **st** B

Una variable de iteración es un entero declarado implícitamente; su alcance es limitado al cuero de la sentencia for-all. El cuerpo del for-all se ejecuta una vez por cada valor de la variable de iteración, comenzando con el valor de la expr_inicial y finalizando con el valor de la expr_final. Si la cláusula opcional such-that (**st**) está presente, la variable de iteración toma sólo los valores para los que la expresión B es verdadera. Si el rango de cuantificadores es vacío, el cuerpo no se ejecuta.

Si hay más de un cuantificador, se separan con comas. En este caso, el cuerpo del **fa** se ejecuta para cada combinación de valores de las variables de iteración, con la variable más a la derecha variando más rápidamente. Asumiendo que cada iteración del cuerpo del **fa** termina, los valores finales de las variables de iteración son uno más que los valores finales especificados en el cuantificador.

Como ejemplo, la siguiente sentencia for-all transpone una matriz *m*:

fa i := 1 **to** n, j := i + 1**to** n → m[i,j] :=: m[j,i] **af**

Se usa una sentencia de swap dentro del cuerpo para intercambiar elementos. El valor inicial de la segunda variable de iteración depende de la primera, para evitar intercambios redundantes.

Como segundo ejemplo, la siguiente sentencia ordena un arreglo de enteros a[1:n] en orden ascendente:

fa i := 1 **to** n, j := i+1 **to** n **st** a[i] > a[j] → a[i] :=: a[j] **af**

Procedures

Un procedure define un patrón parametrizado para una operación. Su forma general es:

procedure p(f_1 : t_1 ; .....; f_n : t_n) **returns** r : t_r
  declaraciones
  sentencias
end

El identificador p es el nombre del procedure. Los f_i son los nombres de los parámetros formales; los t_i son los tipos correspondientes. La parte de returns es una especificación opcional del nombre y el tipo de la variable de retorno. El cuerpo de un procedure contiene declaraciones de variables locales y sentencias que implementan las acciones del procedure.

Un procedure que no tiene una parte de retorno es invocado explícitamente por medio de la sentencia **call**:

call p(e_1, ..., e_n)

Un procedure que tiene parte de retorno se llama *función*. Se invoca explícitamente apareciendo en una expresión, por ejemplo, dentro de una sentencia de asignación:

x : = p(e_1, ..., e_n)

Cuando se invoca un procedure, los parámetros reales son evaluados y luego pasados al procedure por valor (copy in), por valor/resultado (copy in, copy out) o por resultado (copy out). Un parámetro por valor se indica por la palabra clave **val** en la especificación del parámetro formal (es el default). Un parámetro por valor/resultado se indica por **var**, y los parámetros resultado se indican por **res**.

LOGICA, PROPOSICIONES Y PREDICADOS

Consideremos el siguiente problema llamado búsqueda lineal. Dado un arreglo a[1:n], con n>0; un valor x que se sabe que está en a, quizás más de una vez; el problema es computar el índice i de la primera ocurrencia de x en a, es decir, el valor más chico de i tal que a[i]=x. Podemos resolver este problema por el siguiente programa obvio:

var i := 1
do a[i] ≠ x → i := i + 1 **od**

Cómo puedo *probar* que este programa resuelve correctamente el problema? El punto en cualquier prueba es proveer evidencia convincente de la correctitud de alguna sentencia. Para este problema específico, la descripción en lenguaje corriente y el programa son probablemente evidencia convincente para alguien que entiende el lenguaje corriente y programación. Pero con frecuencia las sentencias en lenguaje corriente son ambiguas. También, la mayoría de los programas (especialmente los concurrentes) son largos y complejos. Entonces es necesario tener un marco más riguroso.

Una *lógica de programación* es un sistema formal que soporta la aproximación asercional para desarrollar y analizar programas. Incluye predicados que caracterizan estados de programa y relaciones que caracterizan el efecto de la ejecución del programa. Esta sección resume aspectos relevantes de los sistemas lógicos formales, lógica proposicional, y lógica de predicados, la cual es una extensión de la lógica proposicional. La próxima sección presenta una lógica de programación para programas secuenciales.

Sistemas Lógicos Formales

Cualquier sistema lógico formal consta de reglas definidas en términos de:

- un conjunto de *símbolos*
- un conjunto de *fórmulas* construidas a partir de estos símbolos

- un conjunto de fórmulas distinguidas llamadas *axiomas*, y
- un conjunto de *reglas de inferencia*

Las fórmulas son secuencias bien formadas de símbolos. Los axiomas son fórmulas especiales que *a priori* se asumen verdaderas. Las reglas de inferencia especifican cómo derivar fórmulas verdaderas adicionales a partir de axiomas y otras fórmulas verdaderas.

Las reglas de inferencia tienen la forma:

$$\frac{H_1, H_2, ..., H_n}{C}$$

Las H_i son *hipótesis*; C es una conclusión. Ambos son o fórmulas o representaciones esquemáticas de fórmulas. El significado de una regla de inferencia es que si todas las hipótesis son verdaderas, entonces podemos inferir que la conclusión también lo es.

Una *prueba* en un sistema lógico formal es una secuencia de líneas, cada una de las cuales es un axioma o puede ser derivada de líneas previas por la aplicación de una regla de inferencia. Un *teorema* es cualquier línea en una prueba. Así, los teoremas son axiomas o se obtienen aplicando una regla de inferencia a otros teoremas.

Un sistema lógico formal es una abstracción matemática (colección de símbolos y relaciones entre ellos). Se vuelve interesante cuando las fórmulas representan sentencias sobre algún dominio de discurso y las fórmulas que son teoremas son sentencias verdaderas. Esto requiere dar una interpretación a las fórmulas. Una *interpretación* de una lógica mapea cada fórmula a verdadero o falso. Una lógica es *sound* (fuerte, robusta) con respecto a una interpretación si todos sus axiomas y reglas de inferencia son sound. Un axioma es sound si se mapea a verdadero; una regla de inferencia es sound si su conclusión se mapea a verdadera, asumiendo que todas las hipótesis se mapean a verdadero. Así, si una lógica es sound, todos los teoremas en la lógica son sentencias verdaderas en ese dominio de discurso. En este caso, la interpretación se llama *modelo* para la lógica.

Completitud es el dual de soundness. Una lógica es completa con respecto a una interpretación si toda fórmula que es mapeada a verdadero es un teorema; es decir, la fórmula es probable en la lógica. Una lógica que es sound y completa permite que todas las sentencias exresables en la lógica sean probadas. Si un teorema no puede ser probado en tal lógica, no es el resultado de una debilidad de la lógica.

Desafortunadamente, el dominio de discurso que nos ocupa (las propiedades de los programas concurrentes) no pueden tener una axiomatización sound y completa como un sistema lógico. Esto es porque el comportamiento de un programa incluye aritmética, y un resultado conocido en lógica (teorema de incompletitud de Godel) establece que ningún sistema lógico formal que axiomatiza la aritmética puede ser completo. Pero una lógica que extiende otra lógica puede ser *relativamente completa*, es decir que no introduce fuente de incompletitud más allá de la que se encuentra en la lógica que extiende. Afortunadamente, la completitud relativa es lo suficientemente buena ya que las propiedades aritméticas que emplearemos son verdaderas, aún si no todas pueden ser probadas formalmente.

Proposiciones

La lógica proposicional es una instancia de un sistema lógico formal que formaliza lo que llamamos razonamiento de “sentido común”. Las fórmulas de la lógica son llamadas proposiciones; son sentencias que son verdaderas o falsas. Los axiomas son proposiciones especiales que se asumen verdaderas; por ejemplo, “Es un día soleado implica que es de día” y “Es de día implica que las estrellas no son visibles”. Las reglas de inferencia permiten que se formen nuevas proposiciones verdaderas a partir de las existentes. Por ejemplo, si tuviéramos una regla de transitividad, de las dos sentencias anteriores podríamos concluir que “Es un día soleado implica que las estrellas no son visibles”.

Para ser más precisos, en una lógica proposicional los símbolos proposicionales son:

Constantes Proposicionales: *true* y *false*
Variables Proposicionales: *p*, *q*, *r*, ...
Operadores Proposicionales: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, =

Las fórmulas de la lógica son constantes y variables proposicionales simples, o están construidas usando los operadores proposicionales (conectivos). Hay 5 operadores: negación, conjunción, disyunción, implicación y equivalencia. Sus interpretaciones son las tablas de verdad correspondientes.

Dado un estado *s*, interpretamos una fórmula proposicional *P* como sigue. Primero, reemplazamos cada variable proposicional en *P* por su valor en *s*. Luego se usan las tablas para simplificar el resultado. El orden de precedencia es negación, conjunción y disyunción, implicación y equivalencia.

Una fórmula *P* se *satisface* en un estado si es verdadera en ese estado; *P* es satisfacible si hay algún estado en el cual es satisfecha. La fórmula *P* es *válida* si es satisfacible en cualquier estado. Por ejemplo, $p \vee \neg p$ es válida. Una proposición válida se llama *tautología*.

Una forma de decidir si una fórmula es válida es determinar si su interpretación es verdadera en todo estado posible. Sin embargo, si la fórmula contiene *n* variables proposicionales, esto requiere chequear 2^n casos. Una mejor aproximación es emplear tautologías que permiten que las fórmulas sean simplificadas transformándolas en fórmulas equivalentes.

Hay muchas tautologías dentro de la lógica proposicional; las que más usaremos están listadas a continuación. Son llamadas leyes de equivalencia proposicional ya que permiten que una proposición sea reemplazada por una equivalente.

Ley de Negación: $P = \neg (\neg P)$

Ley del Medio Excluido: $P \vee \neg P = true$

Ley de Contradicción: $P \wedge \neg P = false$

Ley de Implicación: $P \Rightarrow Q = \neg P \vee Q$

Ley de Igualdad: $(P = Q) = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$

Leyes de Or-Simplificación: $P \vee P = P$
$P \vee true = true$
$P \vee false = P$
$P \vee (P \wedge Q) = P$

Leyes de And-Simplificación: $P \wedge P = P$
$P \wedge true = P$
$P \wedge false = false$
$P \wedge (P \vee Q) = P$

Leyes Conmutativas: $(P \wedge Q) = (Q \wedge P)$
$(P \vee Q) = (Q \vee P)$
$(P = Q) = (Q = P)$

Leyes Asociativas: $P \wedge (Q \wedge R) = (P \wedge Q) \wedge R)$
$P \vee (Q \vee R) = (P \vee Q) \vee R)$

Leyes Distributivas: $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$

Leyes de De Morgan: $\neg (P \wedge Q) = \neg P \vee \neg Q$
$\neg (P \vee Q) = \neg P \wedge \neg Q$

Para ilustrar el uso de las leyes de equivalencia, consideremos la siguiente caracterización del problema de la sección crítica. Sean *InA* e *InB* las proposiciones:

InA: Proceso A ejecutando en su SC
InB: Proceso B ejecutando en su SC

Supongamos que sabemos que si A está en su SC, entonces B no está en su SC; es decir que $InA \Rightarrow \neg InB$. Entonces podemos reescribir esta fórmula usando dos de las leyes anteriores:

$InA \Rightarrow \neg InB = (\neg InA \vee \neg InB)$ por Ley de Implicación
$= \neg (InA \wedge InB)$ por Ley de De Morgan

Dado que la equivalencia es transitiva, tenemos:

$InA \Rightarrow \neg InB = \neg (InA \wedge InB)$

Por lo tanto, no es el caso en que los procesos A y B puedan estar en su SC al mismo tiempo.

Otras dos tautologías útiles son:

And-Eliminación: $(P \wedge Q) \Rightarrow P$

Or-Introducción: $P \Rightarrow (P \vee Q)$

No son reglas de equivalencia sino de implicación, que nos permiten concluir que el lado derecho es verdadero si el izquierdo lo es.

Ambas reglas son ejemplos de debilitamiento de una proposición. Por ejemplo, si un estado satisface $P \wedge Q$, entonces satisface P. Así, P es una proposición más débil que $P \wedge Q$ dado que en general más estados satisfacen sólo P que P y Q. Similarmente, $P \vee Q$ es una proposición más débil que P (o que Q). Dado que *false* implica cualquier cosa, *false* es la proposición más fuerte (ningún estado la satisface). Similarmente, *true* es la proposición más débil (cualquier estado la satisface).

Predicados

La lógica proposicional provee las bases para el razonamiento asercional. Sin embargo, por sí misma es demasiado restrictiva ya que las únicas constantes proposicionales son los valores booleanos *true* y *false*. Los lenguajes de programación tienen tipos de datos y valores adicionales tales como enteros y reales. Así, necesitamos una manera de manipular cualquier clase de expresión boolean-valued. Una lógica de predicados extiende una lógica proposicional para soportar esta generalización. Las dos extensiones son las siguientes:

- Cualquier expresión tal como x<y que se mapea a true o false puede usarse en lugar de una variable proposicional
- Se proveen cuantificadores existenciales ($\exists$) y universales ($\forall$) para caracterizar conjuntos de valores

Los símbolos de la lógica de predicados son los de la lógica proposicional más otras variables, operadores relacionales y cuantificadores. Las fórmulas (llamadas *predicados*) son fórmulas proposicionales en las cuales pueden usarse expresiones relacionales y cuantificadas en lugar de variables proposicionales. Para interpretar un predicado en un estado, primero interpretamos cada expresión relacional y cuantificada (obteniendo true o false para cada una) y luego interpretamos la fórmula proposicional resultante.

Una *expresión relacional* contiene dos términos separados por un operador relacional tal como =, $\neq$ o >. Estos operadores están definidos para caracteres, enteros, reales y a veces strings, registros, etc. Introduciremos operadores relacionales adicionales tales como miembro de ($\in$) y subconjunto ($\subset$) según sea necesario.

Una expresión relacionales es true o false, dependiendo de si la relación se da entre los argumentos. Asumimos que cada expresión relacional está bien definida (no hay problemas de tipos).

Cuando trabajamos con conjuntos de valores, con frecuencia queremos asegurar que algunos o todos los valores satisfacen una propiedad (por ej, que todos los elementos de un arreglo son 0). Las *expresiones cuantificadas* proveen un método preciso y compacto.

El *cuantificador existencial* $\exists$ permite asegurar que algún elemento de un conjunto satisface una propiedad. Aparece en expresiones de la forma:

(1.1) $(\exists b_1, ..., b_n : R: P)$

Los b_i se llaman *variables ligadas*; R es una fórmula que especifica el conjunto de valores (rango) de las variables ligadas; P es un predicado. La interpretación de (1.1) es true si P es true para *alguna* combinación de los valores de las variables ligadas. Por ejemplo:

$$(\exists i : 1 \leq i \leq n : a[i] = 0)$$

es true si algún elemento de a[1:n] es 0.

El *cuantificador universal* $\forall$ permite asegurar que todos los elementos de un conjunto satisfacen una propiedad. Aparece en expresiones de la forma:

(1.2) $(\forall b_1, ..., b_n : R: P)$

Nuevamente, las b_i son *variables ligadas*, R especifica sus valores, y P es un predicado. La interpretación de (1.2) es true si P es verdadera para *todas* las combinaciones de los valores de las variables ligadas. Por ejemplo:

$$(\forall i : 1 \leq i \leq n : a[i] = 0)$$

es true si todos los elementos de a[1:n] son 0.

En una expresión cuantificada, el alcance de una variable ligada es la expresión en sí misma.

Una ocurrencia de una variable en un predicado se dice *libre* si (1) no está dentro de un cuantificador o (2) está dentro de un cuantificador y es distinta del nombre de cualquier variable ligada cuyo alcance incluye ese cuantificador. Todas las ocurrencias de variables ligadas son *bound* dentro de su alcance. En los ejemplos anteriores, todas las ocurrencias de a son libres y todas las ocurrencias de i son bound. Sin embargo, en el siguiente predicado, la primera ocurrencia de i es libre pero las otras son bound:

$$i = j \wedge (\exists i, k : i, k > 0 : a[i] = a[k])$$

También, la ocurrencia de j es libre, las ocurrencias de k son bound, y las ocurrencias de a son libres.

Los cuantificadores existencial y universal son duales uno del otro: sólo uno es necesario, aunque ambos son convenientes. En particular, consideremos la siguiente expresión:

$$(\exists B: R: P)$$

Si la interpretación es true en un estado, entonces alguna combinación de los valores de las variables ligadas satisface P. Análogamente, si la interpretación es false, entonces P es false para todas las combinaciones de valores de B. En cualquier caso, la siguiente expresión tiene la misma interpretación:

$$\neg (\forall B: R: \neg P)$$

Una dualidad similar existe en la otra dirección. Las siguientes son algunas leyes (axiomas) que emplearemos:

Leyes de De Morgan para cuantificadores: $(\exists B: R: P) = \neg (\forall B: R: \neg P)$
$(\forall B: R: P) = \neg (\exists B: R: \neg P)$

Ley de Conjunción: $(\forall B: R: P \wedge Q) = (\forall B: R: P) \wedge (\forall B: R: Q)$

Ley de Disyunción: $(\exists B: R: P \vee Q) = (\exists B: R: P) \vee (\exists B: R: Q)$

Leyes de Rango Vacío: $(\forall B: \varnothing: P) = true$
$(\exists B: \varnothing: P) = false$

Para ilustrar el uso de la primera ley de rango vacío, consideremos el siguiente predicado, que asegura que todos los elementos del 1al k de un arreglo a están ordenados ascendentemente:

$$(\forall i : 1 \leq i \leq k-1 : a[i] \leq a[i+1])$$

Si k es 1 (por ejemplo, al comienzo de un programa de ordenación) la expresión es true ya que el rango es el conjunto vacío. En otras palabras, ninguno de los elementos de a es necesariamente ordenados.

El concepto final que emplearemos de la lógica de predicados es la sustitución textual.

(1.3) **Sustitución Textual**: Si ninguna variable en la expresión *e* tiene el mismo nombre que otra variable ligada en el predicado *P*, entonces P_e^x se define como el resultado de sustituir e para cada ocurrencia libre de x en P.

P_e^x se lee “P con x reemplazada por e”. Los nombres de las variables en e no deben conflictuar con variables ligadas.

La definición (1.3) trata con la sustitución de una expresión para una variable en un predicado. Podemos generalizar la definición para permitir sustitución simultánea de varias expresiones para varias variables libres:

(1.4) **Sustitución Simultánea**: Si ninguna variable en las expresiones *e1, ..., en* tiene el mismo nombre que otra variable ligada en el predicado *P*, y si *x1, ...,* xn son identificadores distintos, entonces $P_{e1,.....,en}^{x1,....,xn}$ se define como el resultado de sustituir simultáneamente las e’s para cada ocurrencia de las x’s en P.

Se requiere que las x’s sean distintas en (1.4) pues sino más de una expresión podría ser sustituida por la misma variable. Sin embargo, si dos identificadores x1 y x2 son el mismo, la sustitución simultánea está bien definida si e1 y e2 son sintácticamente la misma expresión. Usaremos esta propiedad en el axioma para la sentencia de swap.

UNA LOGICA DE PROGRAMACION

Una lógica de programación es un sistema lógico formal que facilita hacer precisiones sobre la ejecución de un programa. En esta sección se describe una lógica de programación (*PL*) para las sentencias de programación secuencial descriptas. Posteriormente se extiende esta lógica para incluir procedures y varias construcciones de programación concurrente.

Como cualquier sistema lógico formal, PL contiene símbolos, fórmulas, axiomas y reglas de inferencia. Los símbolos son predicados sentencias del lenguaje de programación. Las fórmulas de PL son *triplas* de las forma:

{ P } S { Q }

P y Q son predicados, y S es una sentencia simple o compuesta.

En P y Q, las variables libres son variables de programa y variables lógicas. Las *variables de programa* son introducidas en las declaraciones. Las *variables lógicas* son variables especiales que sirven para mantener valores arbitrarios; aparecen sólo en predicados, no en sentencias de programa.

Dado que el propósito de PL es facilitar la prueba de propiedades de la ejecución de programas, la interpretación de una tripla caracteriza la relación entre los predicados P y Q y el efecto de ejecutar la sentencia S.

(1.5) **Interpretación de una tripla**. Sea que cada variable lógica tiene algún valor del tipo correcto. Luego, la interpretación de una tripla { P } S { Q } es true si, para cualquier ejecución de S que comienza en un estado que satisface P y la ejecución de S termina, el estado resultante satisface Q.

La interpretación es llamada *corrección parcial*, la cual es una propiedad de seguridad. Dice que, si el estado inicial del programa satisface P, entonces el estado final en cualquier historia finita resultante de ejecutar S, va a satisfacer Q. La propiedad de vida relacionada es la *corrección total*, que es corrección parcial más terminación; es decir que todas las historias son finitas.

En una tripla, P y Q se llaman *aserciones*, ya que afirman que el estado de programa debe satisfacer el predicado para que la interpretación de la tripla sea true. Así, una aserción caracteriza un estado de programa aceptable. El predicado P es llamado *precondición* de S (se anota *pre(S)*; caracteriza la condición que el estado debe satisfacer antes de que comience la ejecución de S. El predicado Q se llama *postcondición* de S (*post(S)*); caracteriza el estado que resulta de ejecutar S, si S termina. Dos aserciones especiales son *true*, que caracteriza todos los estados de programa, y *false*, que caracteriza a ningún estado.

Para que una interpretación sea un modelo para PL, los axiomas y reglas de inferencia de PL deben ser sound con respecto a (1.5). Esto asegurará que todos los teoremas probables en PL son sound. Por ejemplo, la siguiente tripla debería ser un teorema:

{ x=0 } x := x + 1 { x = 1 }

Pero, la siguiente no sería un teorema, ya que asignar un valor a x no puede "milagrosamente" setear y a 1:

{ x=0 } x := x + 1 { y = 1 }

Además de ser sound, la lógica debería ser (relativamente) completa para que todas las triplas que son true sean de hecho probables como teoremas.

Axiomas

Se presentan axiomas y reglas de inferencia de PL y justificaciones informales de su soundness y completitud relativa. Asumiremos que la evaluación de expresiones no causa efectos laterales, es decir, ninguna variable cambia de valor.

La sentencia skip no cambia ninguna variable. Luego, si el predicado P es true antes de ejecutarse **skip**, sigue siendo true cuando **skip** termina:

(1.6) **Axioma de Skip**: { P } **skip** { P }

Una sentencia de asignación asigna un valor *e* a una variable *x* y en general cambia el estado del programa. Parecería que el axioma para la asignación debiera empezar con alguna precondición *P*, y que la postcondición debiera ser *P* más un predicado para indicar que *x* ahora tiene el valor *e*. Pero, resulta un axioma más simple si vamos en la otra dirección. Asumamos que la postcondición de una asignación es satisfacer *P*. Entonces, qué debe ser true antes de la asignación? Primero, una asignación cambia solo la variable destino *x*, entonces todas las otras variables tienen el mismo valor antes y después. Segundo, *x* tiene un nuevo valor *e*, y así toda relación que pertenece a *x* que es true después de asignar tiene que haberlo sido antes de reemplazar *x* por *e*. La sustitución textual (1.3) hace exactamente esta transformación:

(1.7) **Axioma de Asignación**: $\{P_e^x\}$ x := e { P }

Para ilustrar el uso de este axioma, consideremos la siguiente tripla:

{ true } x := 5 { x = 5 }

Es un teorema dado que:

$(x = 5)_5^x$ = (5 = 5) = true

Esto indica que comenzar en cualquier estado y asignar un valor a una variable le da ese valor a la variable. Como segundo ejemplo, consideremos la tripla:

{ y = 1 } x := 5 { y = 1 $\wedge$ x = 5 }

Esto también es un teorema, ya que:

$(y = 1 \wedge x = 5)_5^x$ = (y = 1 $\wedge$ 5 = 5) = (y = 1 $\wedge$ true) = (y = 1)

Esto ilustra que las relaciones para las variables que no son asignadas no son afectadas por una asignación.

Desafortunadamente, (1.7) no es sound para asignaciones a elementos de un arreglo o campos de registro. Consideremos la siguiente tripla:

$\{ P_8^{a[3]} \}$ a[3] := 8 { P: i=3 $\wedge$ a[i] = 6 }

Claramente, la postcondición P no debería ser satisfecha y la interpretación de la tripla sería falsa. Pero cuál es la precondición que resulta del axioma de asignación? Debería ser:

i = 3 $\wedge$ 8 = 6

Esto es falso, pero para alcanzar esta conclusión tenemos que darnos cuenta que a[i] y a[3] son el mismo elemento. No podemos hacerlo sólo con sustitución textual.

Podemos ver un arreglo como una colección de variables independientes a[1], a[2], etc. Alternativamente, podemos ver un arreglo como una función (parcial) de los valores suscriptos a los elementos del arreglo. Con esta segunda visión, un arreglo se convierte en una variable simple que contiene una función.

Sea *a* un arreglo unidimensional, y sean *i* y *e* expresiones cuyos tipos matchean los del rango y el tipo base de *a*, respectivamente. Denotemos con (*a; i : e*) el arreglo (función) cuyo valor es el mismo que *a* para todos los índices excepto *i*, donde su valor es *e*:

(1.8) $(a; i : e)[j] = \begin{cases} a[j] & \text{si } i \neq j \\ e & \text{si } i = j \end{cases}$

Con esto, a[i] = e es simplemente una abreviación para a := (a; i : e); es decir, reemplazamos a por una nueva función que es la misma que la vieja, excepto quizás en el lugar i. Dado que a es ahora una variable simple, se aplica (1.7). Podemos manejar las asignaciones a arreglos multidimensionales y registros de manera similar.

Para ilustrar la notación funcional para arreglos, consideremos nuevamente la tripla:

$\{ P_8^{a[3]} \}$ a[3] := 8 { P: i=3 $\wedge$ a[i] = 6 }

Reescribiendo la asignación como a := (a; 3 : 8) y sustituyendo en P queda:

$P^{a}_{(a;3:8)}$ = (i=3 $\wedge$ (a; 3 : 8)[i] = 6)

De la definición (1.8) junto al hecho de que i=3, la parte derecha se simplifica a:

(i=3 $\wedge$ 8=6) = false

Esta es una interpretación sound ya que a[3] no puede ser igual a 6 luego de que se le asigna 8. Lo que ha ocurrido es que, reescribiendo la asignación al arreglo, toda la información relevante acerca de a fue transportada en la sustitución textual.

La sentencia de swap intercambia los valores de dos variables v1 y v2. El efecto es asignar simultáneamente v1 a v2 y v2 a v1. Luego, el axioma para el swap generaliza (1.7) empleando sustitución simultánea (1.4).

(1.9) **Axioma de Swap**: $\{ P^{v1,v2}_{v2,v1} \}$ v1 :=: v2 { P }

Como ejemplo, el siguiente teorema se deriva directamente del axioma de swap:

{ x=X $\wedge$ y=Y } x :=: y { x=Y $\wedge$ y=X }

X e Y son variables lógicas. Manejamos swapping de elementos de arreglos o campos de registros viéndolos como funciones parciales.

Reglas de inferencia

El estado de programa cambia como resultado de ejecutar sentencias de asignación y swap. Luego, los axiomas para estos elementos usan sustitución textual para introducir nuevos valores en los predicados. Las reglas de inferencia de PL permiten que los teoremas resultantes de estos axiomas sean manipulados y combinados. Hay una regla de inferencia para cada una de las sentencias que afectan el flujo de control en un programa secuencial: composición, alternativa e iteración. Hay una regla de inferencia adicional que usamos para conectar triplas una con otra.

La primera regla de inferencia, la Regla de Consecuencia, nos permite manipular predicados en triplas. Consideremos la siguiente tripla:

(1.10) { x=3 } x := 5 { x=5 }

Claramente sería un teorema dado que *x* vale 5 luego de la asignación, independientemente del valor que tuviera antes. Sin embargo, esto no se desprende directamente del axioma de asignación (1.7). Como ya dijimos, lo que se desprende del axioma es el teorema:

(1.11) { true } x := 5 { x=5 }

Recordemos que true caracteriza cualquier estado de programa, incluyendo x=3; en particular, (x=3) $\Rightarrow$ true. La Regla de Consecuencia nos permite hacer esta conexión y concluir a partir de la validez de (1.11) que (1.10) también es válida.

(1.12) **Regla de Consecuencia**: $\dfrac{P' \Rightarrow P,\ \{P\}\ S\ \{Q\},\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$

La regla de consecuencia nos permite fortalecer una precondición, debilitarla o ambas.

Una sentencia compuesta ejecuta la sentencia S1, luego S2. La Regla de Composición nos permite combinar triplas válidas concernientes a S1 y S2.

(1.12) **Regla de Composición**: $\dfrac{\{P\}\ S1\ \{Q\},\ \{Q\}\ S2\ \{R\}}{\{P\}\ S1;\ S2\ \{R\}}$

Por ejemplo, consideremos la sentencia compuesta:

x := 1; y := 2

A partir del axioma de asignación, los siguientes son teoremas:

{ true } x := 1 { x=1 }
{ x=1 } y := 2 { x=1 $\wedge$ y=2 }

Por la regla de composición, el siguiente también es un teorema:

{ true } x := 1; y := 2 { x=1 $\wedge$ y=2 }

Una sentencia alternativa (**if**) contiene n sentencias guardadas:

IF: **if** $B_1 \rightarrow S_1$ □ ..... □ $B_n \rightarrow S_n$ **fi**

Recordemos que asumimos en PL que la evaluación guardada no tiene efectos laterales, es decir que B_i no contiene llamadas a funciones que cambian parámetros resultado o variables globales. Por lo tanto, ninguna variable de programa puede cambiar el estado como resultado de evaluar una guarda. Con esta suposición, si ninguna guarda es true, la ejecución de IF es equivalente a **skip**. Por otra parte, si al menos una guarda es true, entonces una de las sentencias guardadas será ejecutada, con la elección no determinística si más de una es true.

Supongamos que la precondición de IF es P y la postcondición deseada es Q. Luego, si P es true y todas las B_i son falsas cuando el IF es ejecutado, Q es true ya que en este caso If es equivalente a **skip**. Sin embargo, si alguna B_i es true y S_i es elegida para ejecutarse, entonces el siguiente es un teorema:

{ P$\wedge B_i$ } S_i { Q }

Necesitamos teoremas para cada sentencia guardada ya que no podemos saber cuál será ejecutada. Poniendo todo esto junto, tenemos la siguiente regla:

(1.14) **Regla de Alternativa**: $P \wedge \neg(B_1 \vee ..... \vee B_n) \Rightarrow Q$

$$\frac{\{ P \wedge B_i \}\ S_i\ \{ Q \}\ ,\ 1 \le i \le n}{\{ P \}\ \text{IF}\ \{ Q \}}$$

Como ejemplo del uso de esta regla, consideremos el siguiente programa:

if $x \ge y \rightarrow m := x$ □ $y \ge x \rightarrow m := y$ **fi**

Esto asigna a m el máximo de x e y. Supongamos inicialmente que x=X e y=Y. Luego, la siguiente tripla sería un teorema:

(1.15) { P: $x=X \wedge y=Y$ }
if $x \ge y \rightarrow m := x$ □ $y \ge x \rightarrow m' := y$ **fi**
{ $P \wedge MAX$ }

Donde MAX es el predicado:

MAX: $(m=X \wedge X \ge Y) \vee (m=Y \wedge Y \ge X)$

Para concluir que (1.15) es un teorema, debemos satisfacer la hipótesis de la Regla de Alternativa. La primera hipótesis es trivial de satisfacer para esta alternativa ya que al menos una guarda es true. Para satisfacer la segunda hipótesis, tenemos que considerar cada una de las sentencias guardadas. Para la primera, aplicamos el axioma de asignación a la sentencia m := x con postcondición $P \wedge MAX$ y tenemos el teorema:

$\{ R: P \wedge ((x=X \wedge X \ge Y) \vee (x=Y \wedge Y \ge X)) \}\ m := x\ \{ P \wedge MAX \}$

Si la primera sentencia guardada es seleccionada, entonces el estado debe satisfacer:

$(P \wedge x \ge y) = (P \wedge X \ge Y)$

Dado que esto implica que R es true, podemos usar la Regla de Consecuencia para tener el teorema:

$\{ P \wedge x \ge y \}\ m := x\ \{ P \wedge MAX \}$

Un teorema similar se tiene para la segunda sentencia guardada. Así, podemos usar la regla de alternativa para inferir que (1.15) es un teorema de PL.

Con frecuencia usamos la regla de consecuencia para construir triplas que satisfacen la hipótesis de la regla de alternativa. Arriba lo usamos para demostrar que $P \wedge B_i \Rightarrow pre(S_i)$. Alternativamente, podemos usarla con las postcondiciones de las S_i. Supongamos que las siguientes son triplas válidas:

$\{ P \wedge B_i \}\ S_i\ \{ Q_i \},\ 1 \le i \le n$

Esto es, cada branch de una sentencia alternativa produce una condición posiblemente distinta. Dado que podemos debilitar cada una de tales postcondiciones a la disyunción de todas las Q_i, podemos usar la regla de consecuencia para concluir que lo siguiente es válido:

$\{ P \wedge B_i \}\ S_i\ \{ Q_1 \vee ..... \vee Q_n \},\ 1 \le i \le n$

Por lo tanto, podemos usar la disyunción de las Q_i como postcondición del **if**, asumiendo que también satisfacemos la primera hipótesis de la regla de alternativa.

Ahora consideremos la sentencia iterativa. Recordemos que la ejecución del **do** difiere del **if** en lo siguiente: la selección y ejecución de una sentencia guardada se repite hasta que todas las guardas son falsas. Así, una sentencia **do** puede iterar un número arbitrario de veces, aún cero. Por esto, la regla de inferencia se basa en un *loop invariant*: un predicado (aserción) *I* que se mantiene antes y después de cada iteración del loop. Sea DO la sentencia **do**:

DO: **do** $B_1 \rightarrow S_1$ □ ..... □ $B_n \rightarrow S_n$ **od**

Supongamos que el predicado *I* es true antes de la ejecución de DO y luego de cada iteración. Entonces, si S_i es seleccionada para ejecución, pre(S_i) satisface *I* y B_i, y post(S_i) debe satisfacer *I*. La ejecución de DO termina cuando todas las guardas son falsas; luego, post(DO) satisface esto e *I*. Estas observaciones dan la siguiente regla de inferencia:

(1.16) **Regla Iterativa**: $$\frac{\{ I \wedge B_i \}\ S_i\ \{ I \},\ 1 \le i \le n}{\{ I \}\ DO\ \{ I \wedge \neg (B_1 \vee ....\vee B_n) \}}$$

La clave para usar la regla iterativa es el invariante. Como ejemplo, consideremos el siguiente programa, que computa el factorial de un entero *n*, asumiendo n > 0:

var fact := 1; i := 1`
do i ≠ n → i := i + 1; fact := fact * i **od**

Antes y después de cada iteración, fact contiene el factorial de i. Por lo tanto, la siguiente aserción es un loop invariant:

fact = i! ∧ 1 ≤ i ≤ n

Cuando el loop termina, tanto el invariante como la negación de la guarda del loop son true:

fact = i! ∧ i=n

Por lo tanto, el programa computa correctamente el factorial de n. La sentencia for-all es la que nos queda. Recordemos que **fa** es una abreviación para el uso especial del **do**. Luego, podemos trasladar un programa que contiene **fa** en uno equivalente que contiene **do**, luego usar la regla iterativa para desarrollar una prueba de corrección parcial del programa trasladado.

PRUEBAS EN PL

Dado que la lógica de programación PL es un sistema lógico formal, una prueba consiste en una secuencia de líneas. Cada línea es una instancia de un axioma o se deduce de líneas previas por aplicación de una regla de inferencia. Los axiomas y reglas de inferencia de PL son los que vimos en la sección anterior más algunos de las lógicas proposicional y de predicados.

Para ilustrar una prueba completa en PL consideremos nuevamente el problema de búsqueda lineal. Se tiene un arreglo a[1:n] para algún positivo n. También se tiene un valor x que es un elemento de a. El problema es encontrar el índice de la primera ocurrencia de x en a. Más precisamente, el estado inicial se asume que satisface el predicado:

P: n > 0 ∧ (∃j: 1 ≤ j ≤ n: a[j] = x)

Y el estado final del programa debe satisfacer:

LS: a[i] = x ∧ (∀ j : 1 ≤ j < i: a[j] ≠ x)

La primera parte de LS dice que i es un índice tal que a[i] = x; la segunda parte dice que ningún índice más chico satisface esta propiedad. Está implícito en el enunciado del problema que n, a y x no deberían cambiar. Podríamos especificar esto formalmente incluyendo el siguiente predicado en P y LS; en el predicado, N, A, y X son variables lógicas:

n = N ∧ (∀ i: 1 ≤ i ≤ n: a[i] = A[i]) ∧ x = X

Por simplicidad lo omitiremos en la prueba.

A continuación se dará una prueba completa de que la siguiente tripla es válida y por lo tanto que el programa es correcto parcialmente:

{ P }
i := 1
do a[i] ≠ x → i := i + 1 **od**
{ LS }

Construimos la prueba considerando primero el efecto de la primera asignación, luego trabajando dentro del loop para considerar el efecto del loop, y finalmente considerando el efecto de la sentencia compuesta. El loop invariant *I* es el segundo conjuntor de LS; aparece en el paso 4 de la pureba.

1. { P $\wedge$ 1=1 } por Axioma de Asignación
   i := 1
   { P $\wedge$ i=1 }

2. (P $\wedge$ 1=1) = P por Lógica de Predicados

3. { P } por regla de consecuencia con 1 y 2
   i := 1
   { P $\wedge$ i=1 }

4. { P $\wedge$ ($\forall$ j : 1 $\leq$ j < i+1: a[j] $\neq$ x) } por axioma de asignación
   i := i + 1
   { I: P $\wedge$ ($\forall$ j : 1 $\leq$ j < i: a[j] $\neq$ x) }

5. (I $\wedge$ a[i] $\neq$ x) = por Lógica de Predicados
   (P $\wedge$ ($\forall$ j : 1 $\leq$ j < i+1: a[j] $\neq$ x))

6. { I $\wedge$ a[i] $\neq$ x } por regla de consecuencia con 4 y 5
   i := i + 1
   { I }

7. { I } por regla iterativa con 6
   do a[i] $\neq$ x $\rightarrow$ i := i + 1 **od**
   { I $\wedge$ a[i] = x }

8. (P $\wedge$ i=1) $\Rightarrow$ I por Lógica de predicados

9. { P } por regla de composición con 7 y 8
   i := 1
   do a[i] $\neq$ x $\rightarrow$ i := i + 1 **od**
   { I $\wedge$ a[i] = x }

10. (I $\wedge$ a[i] = x) $\Rightarrow$ LS por Lógica de Predicados

11. { P } por regla de consecuencia con 9 y 10
    i := 1
    do a[i] $\neq$ x $\rightarrow$ i := i + 1 **od**
    { LS }

Dado que la tripla en el paso 11 es un teorema, el programa de búsqueda lineal es parcialmente correcto. Dado que P postula la existencia de algún x tal que a[i] = x, el loop termina. Así, el programa también satisface la propiedad de corrección total.

Proof Outlines

Como muestra el ejemplo anterior, es tedioso construir una prueba formal en PL (o cualquier sistema lógico formal). La prueba tiene la virtud de que cada línea puede ser chequeada mecánicamente. Sin embargo, la forma de la prueba la hace difícil de leer.

Un *proof outline* (a veces llamado *programa comentado*) provee una manera compacta en la cual presentar el esbozo de una prueba. Consiste de las sentencias de un programa con aserciones intercaladas. Un *complete proof outline* contiene al menos una aserción antes y después de cada sentencia. Por ejemplo, la siguiente es un proof outline completo para el programa de búsqueda lineal:

```
{ P: n > 0 ∧ ( ∃ j : 1 ≤ j ≤ n : a[j] = x ) }
i := 1
{ P ∧ i = 1 }
{ I: P ∧ ( ∀ j : 1 ≤ j < i: a[j] ≠ x ) }
do a[i] ≠ x → { I ∧ a[i] ≠ x }
                i := i + 1
```

{ I }
od
{ I $\wedge$ a[i] = x }
{ LS: x=a[i] $\wedge$ ($\forall$ j : 1 $\leq$ j < i: a[j] $\neq$ x) }

La correspondencia entre la ejecución del programa y un proof outline es que, cuando el control del programa está al comienzo de una sentencia, el estado de programa satisface la aserción correspondiente. Si el programa termina, la aserción final eventualmente se vuelve true. Dado que el programa de búsqueda lineal termina, eventualmente el control de programa está en el final, y el estado satisface LS.

Un proof outline completo incluye aplicaciones de los axiomas y reglas de inferencia para cada sentencia en el proof outline. En particular, representa los pasos en una prueba formal de la siguiente manera:

- Cada sentencia de **skip**, asignación o swap junto con sus pre y postcondiciones forma una tripla que representa una aplicación del axioma correspondiente.

- Las aserciones antes de la primera y después de la última de una secuencia de sentencias, junto con las sentencias intervinientes (pero no las aserciones intervinientes), forman una tripla que representa una aplicación de la regla de composición.

- Una sentencia alternativa junto con sus pre y postcondiciones representa una aplicación de la regla de alternativa, con las hipótesis representadas por las aserciones y sentencias en las sentencias guardadas.

- Una sentencia iterativa junto con sus pre y postcondiciones representa una aplicación de la regla iterativa; nuevamente las hipótesis de la regla son representadas por las aserciones y sentencias en las sentencias guardadas.

- Finalmente, las aserciones adyacentes representan aplicaciones de la regla de consecuencia.

Las aserciones en proof outlines pueden ser vistas como comentarios precisos en un lenguaje de programación. Caracterizan exactamente qué es true del estado en varios puntos del programa. Así como no siempre es necesario poner comentarios en cada sentencia del programa, no siempre es necesario poner una aserción antes de cada sentencia en un proof outline. Así, generalmente pondremos aserciones sólo en puntos críticos donde ayuden a proveer “evidencia convincente” de que un proof outline de hecho representa una prueba. Como mínimo, los puntos críticos incluyen el comienzo y final de un programa y el comienzo de cada loop.

Equivalencia y simulación

Ocasionalmente, es interesante saber si dos programas son intercambiables, es decir, si computan exactamente los mismos resultados. En otras oportunidades interesa saber si un programa simula a otro, es decir, el primero computa al menos todos los resultados del segundo.

En PL, dos programas se dice que son parcialmente equivalentes si cada vez que comienzan en el mismo estado inicial, terminan en el mismo estado final, asumiendo que ambos terminan.

(1.17) **Equivalencia Parcial**. Las listas de sentencias S1 y S2 son parcialmente equivalentes si, para todos los predicados P y Q, { P } S1 { Q } es un teorema si y solo si { P } S2 { Q } es un teorema.

Esto se llama equivalencia parcial dado que PL es una lógica para probar solo propiedades de correctitud parcial. Si uno de S1 y S2 termina pero el otro no, deberían ser equivalentes de acuerdo a la definición anterior cuando en realidad no lo son. En un programa secuencial, si S1 y S2 son parcialmente equivalentes y ambos terminan, son intercambiables. Lo mismo no es necesariamente true en un programa concurrente debido a la potencial interferencia entre procesos.

Como ejemplo, los siguientes dos programas son parcialmente equivalentes:

S1: v1 :=: v2 S2: v2 :=: v1

Esto es porque, para cualquier postcondición P, la sustitución simultánea da la misma precondición independiente del orden en el cual aparecen las variables. Las siguientes sentencias también son parcialmente equivalentes:

S1: x :=1 ; y := 1 S2: y := 1 ; x := 1

En PL, un programa se dice que simula a otro si, cada vez que ambos empiezan en el mismo estado inicial y terminan, el estado final del primer programa satisface todas las aserciones que se aplican al estado final del segundo. Esencialmente, la simulación es equivalencia parcial en una dirección:

(1.17) **Simulación**. La lista de sentencias S1 simula a S2 si, para todos los predicados P y Q, { P } S1 { Q } es un cada vez que { P } S2 { Q } es un teorema.

Por ejemplo, las siguientes sentencias simulan x :=: y

t := x ; x := y ; y := t

Estas sentencias no son equivalentes a x :=: y, ya que la simulación contiene una variable adicional t que podría ser usada en el programa circundante. Aunque la implementación de una sentencia swap en la mayoría de las máquinas requeriría usar una variable temporaria, esa variable no sería visible al programador y por lo tanto no podría ser usada en otro lugar del programa.

DERIVACION DE PROGRAMAS

Los ejemplos de la sección previa mostraban cómo usar PL para construir una prueba de corrección parcial *a posteriori*. Esto es importante, pero con frecuencia un programador tiene un objetivo (postcondición) y una suposición inicial (precondición) y se pregunta cómo construir un programa que llegue al objetivo bajo las suposiciones establecidas. Además, típicamente se espera que el programa termina. PL no provee una guía de cómo demostrar esto.

Esta sección presenta un método de programación sistemático (un cálculo de programación) para construir programas secuenciales totalmente correctos. El método se basa en ver las sentencias como *transformadores de predicados*: funciones que mapean predicados en predicados. Involucra desarrollar un programa y su proof outline en conjunto, con el cálculo y el proof outline guiando la derivación del programa.

Precondiciones Weakest

La programación es una actividad “goal-directed”. Los programadores siempre tienen algún resultado que tratan de obtener. Supongamos que el objetivo de un programa S es terminar en un estado que satisface el predicado Q. La precondición weakest *wp* es un transformador de predicado que mapea un objetivo Q en un predicado *wp(S,Q)* de acuerdo a la siguiente definición:

(1.19) **Precondición Weakest**. La precondición weakest de la lista de sentencias S y el predicado Q, denotado *wp(S,Q)* es un predicado que caracteriza el mayor conjunto de estados tal que, si la ejecución de S comenzó en cualquier estado que satisface *wp(S,Q)*, entonces se garantiza que la ejecución termina en un estado que satisface Q.

La relación *wp* es llamada la precondición *weakest* dado que caracteriza el mayor conjunto de estados que llevan a un programa totalmente correcto.

Las precondiciones weakest están muy relacionadas a las triplas en PL. De la definición de *wp*, { *wp(S,Q)* } S { Q } es un teorema de PL. Esto significa que:

(1.20) **Relación entre** *wp* **y** *PL*. Si P $\Rightarrow$ *wp(S,Q)*, entonces { P } S { Q } es un teorema de PL

La diferencia esencial entre *wp* y *PL* es que *wp* requiere terminación, mientras *PL* no. Esta diferencia se reduce a requerir que todos los loops terminen dado que los loops son las únicas sentencias no terminantes en nuestra notación.

Se deducen varias leyes útiles directamente de la definición de *wp*. Primero, un programa S no puede terminar en un estado que satisface false dado que no hay tal estado. Así,

(1.21) **Ley del milagro excluido**. *wp(S,false) = false*

Por otro lado, todos los estados satisfacen true. Así, *wp(S,true)* caracteriza todos los estados para los cuales S se garantiza que termina, independiente del resultado final producido por S.

Supongamos que S comienza en un estado que satisface tanto *wp(S,Q)* y *wp(S,R)*. Luego por (1.20), S terminará en un estado que satisface Q $\wedge$ R. Además, nuevamente por (1.20), un estado que satisface *wp(S,Q$\wedge$R)* satisface tanto *wp(S,Q)* y *wp(S,R)*. Así tenemos:

(1.22) **Ley Distributiva de la Conjunción**. *wp(S,Q)* $\wedge$ *wp(S,R) = wp(S,Q$\wedge$R)*

Consideremos la disyunción. Supongamos que un programa comienza en un estado que satisface *wp(S,Q)* o *wp(S,R)*. Luego por (1.20), S terminará en un estado que satisface Q$\vee$R, por lo tanto:

(1.23) **Ley Distributiva de la Disyunción**. *wp(S,Q)* $\vee$ *wp(S,R)* $\Rightarrow$ *wp(S,Q$\vee$R)*

Sin embargo, la implicación en (1.23) no puede ser reemplazada por igualdad ya que **if** y **do** son sentencias no determinísticas. Para ver por qué, consideremos el siguiente programa, que simula echar una moneda al aire:

```
flip: if true → outcome := HEADS
      □□ true → outcome := TAILS
      fi
```

Dado que ambas guardas en flip son true, cualquier sentencia guardada puede ser elegida. Así, no hay un estado de comienzo que garantiza un valor final particular para outcome. En particular,

wp(flip, outcome=HEADS) = wp(flip, outcome=TAILS) = false

Por otra parte, una de las sentencias guardadas será ejecutada, con lo cual outcome será o HEADS o TAILS cuando flip termina, cualquiera sea el estado inicial. Por lo tanto:

wp(flip, outcome=HEADS $\vee$ outcome=TAILS) = true

Aunque este ejemplo demuestra que la disyunción es en general distributiva en solo una dirección, para sentencias determinísticas la implicación en (1.23) puede ser transformada en igualdad:

(1.23) **Ley Distributiva de la Disyunción Determinística**. Para S determinística, *wp(S,Q)* $\vee$ *wp(S,R) = wp(S,Q$\vee$R)*

Esta ley vale para lenguajes de programación secuenciales, tal como Pascal, que no contienen sentencias no determinísticas.

Precondiciones weakest de sentencias

Esta sección presenta reglas para computar *wp* para las sentencias secuenciales ya introducidas. Hay una regla para cada clase de sentencia. Dado que *wp* está altamente relacionado con PL, la mayoría de las reglas son bastante similares a los axiomas o reglas de inferencia correspondientes de PL.

La sentencia **skip** siempre termina y no cambia ninguna variable lógica o de programa. Así:

(1.25) *wp*(**skip**,Q) = Q

Una sentencia de asignación termina si la expresión y la variable destino están bien definidas. Asumimos esto true. Si la ejecución de x := e debe terminar en un estado que satisface Q, debe comenzar en un estado en el cual cada variable excepto x tiene el mismo valor y x es reemplazada por e. Como en el axioma de asignación, esta es exactamente la transformación provista por la sustitución textual:

(1.26) $wp(x:=e,Q) = Q_e^x$

En este caso, *wp* y la precondición para el axioma de asignación son idénticos. Similarmente, *wp* para una sentencia swap es la misma que la precondición en el axioma de swap.

Una secuencia de sentencias S1 y S2 termina en un estado que satisface Q si S2 termina en un estado que satisface Q. Esto requiere que la ejecución de S2 comience en un estado que satisfaga *wp*(S2,Q), el cual debe ser el estado en que termina S1. Así, la composición de sentencias lleva a la composición de *wp*:

(1.27) $wp(S1;S2,Q) = wp(S1,wp(S2,Q))$

Una sentencia **if** termina si todas las expresiones están bien definidas y la sentencia elegida S_i termina. Sea IF la sentencia:

IF: **if** $B_1 \rightarrow S_1$ □ ..... □ $B_n \rightarrow S_n$ **fi**

Si ninguna guarda es true, entonces ejecutar IF es lo mismo que ejecutar **skip** ya que se asume que la evaluación de expresiones no tiene efectos laterales. Si S_i se ejecuta, entonces la guarda B_i debe haber sido true; para asegurar terminación en un estado que satisface Q, debe ser el caso en que $B_i \Rightarrow wp(S_i,Q)$; es decir, o B es falsa o la ejecución de S termina en un estado que satisface Q. Lo mismo se requiere para todas las sentencias guardadas. Entonces:

(1.28) $wp(IF,Q) = \neg(B_1 \vee ..... \vee B_n) \Rightarrow Q \wedge (B_1 \Rightarrow wp(S_1,Q) \wedge .... \wedge B_n \Rightarrow wp(S_n,Q))$

Como ejemplo, consideremos el programa y postcondición para computar el máximo de x e y dado en (1.15):

if $x \geq y \rightarrow m := x$ □ $y \geq x \rightarrow m := y$ **fi**
{ Q: $x=X \wedge y=Y \wedge ((m=x \wedge X \geq Y) \vee (m=Y \wedge Y \geq X))$ }

Aplicando la definición (1.28) a esta sentencia y predicado (usando (1.26) para computar *wp* de la asignación) se tiene:

$\neg(x \geq y \vee y \geq x) \Rightarrow Q \wedge$
$(x \geq y \Rightarrow (x=X \wedge y=Y \wedge ((x=X \wedge X \geq Y) \vee (x=Y \wedge Y \geq X)))) \wedge$
$(y \geq x \Rightarrow (x=X \wedge y=Y \wedge ((y=X \wedge X \geq Y) \vee (y=Y \wedge Y \geq X))))$

Dado que al menos una de las guardas es true, la primera línea se simplifica a true y por lo tanto puede ser ignorada. Reescribiendo las implicaciones usando la Ley de Implicación, la expresión anterior se simplifica a:

$((x<y) \vee (x=X \wedge y=Y \wedge X \geq Y)) \wedge$
$((y<x) \vee (x=X \wedge y=Y \wedge Y \geq X))$

Usando lógica de predicados y proposicional, esto se simplifica a:

$x=X \wedge y=Y$

Esta es exactamente la precondición P en la tripla de (1.15). Esto nuevamente muestra la dualidad entre precondiciones weakest y teoremas en PL.

La sentencia **do** tiene la precondición weakest más complicada ya que es la única sentencia que podría no terminar. Sea DO una sentencia **do**:

(1.29) DO: **do** $B_1 \rightarrow S_1$ □ ..... □ $B_n \rightarrow S_n$ **od**

Y sea BB el predicado:

(1.30) BB: $B_1 \vee ..... \vee B_n$

Esto es, BB es true si alguna guarda es true, y BB es false en otro caso. Podemos reescribir DO en términos de IF como sigue:

DO: **do** BB $\rightarrow$ IF: **if** $B_1 \rightarrow S_1$ □ ..... □ $B_n \rightarrow S_n$ **fi od**

Ahora sea $H_k(Q)$ un predicado que caracteriza todos los estados desde los cuales la ejecución de DO lleva en *k* o menos iteraciones a terminación en un estado que satisface Q. En particular,

$H_0(Q) = \neg BB \wedge Q$
$H_k(Q) = H_0(Q) \vee wp(IF, H_{k-1}(Q))$

La ejecución del DO termina si realiza sólo un número finito de iteraciones. Así, la precondición weakest de DO es:

(1.31) $wp(DO,Q) = (\exists\ k : 0 \leq k: H_k(Q))$

Desafortunadamente, la definición (1.31) no provee ninguna guía para computar la precondición weakest de una sentencia **do**. Además, la relación entre (1.31) y la correspondiente Regla Iterativa (1.16) de PL es mucho menos clara que para cualquier otra sentencia. Por ejemplo, la regla iterativa emplea un loop invariant I que no aparece en (1.31). Sin embargo, la siguiente definición caracteriza la relación; también provee una técnica para establecer que un loop termina.

(1.32) **Relación entre** $wp(DO,Q)$ **y la Regla Iterativa**. Sea DO una sentencia **do** como se definió en (1.29), y represente BB la disyunción de las guardas como se definió en (1.30). Además, supongamos que *I* es un predicado y *bound* es una expresión entera cuyo valor es no negativo. Entonces $I \Rightarrow wp(DO, I \wedge \neg BB)$ si las siguientes condiciones son true:

(1) $(I \wedge B_i) \Rightarrow wp(S_i, I), 1 \leq i \leq n$
(2) $(I \wedge BB) \Rightarrow bound > 0$
(3) $(I \wedge B_i \wedge bound = BOUND) \Rightarrow wp(S_i, bound < BOUND), 1 \leq i \leq n$

La primera condición afirma que *I* es invariante con respecto a la ejecución de una sentencia guardada. La segunda y tercera tratan con la terminación. En ellas, *bound* es una expresión (llamada *bounding expression*) cuyo rango son los enteros no negativos. La segunda condición en (1.32) dice que el valor de *bound* es positivo si alguna guarda es true. La tercera dice que cada iteración decrementa el valor de *bound*. Dado que el valor de *bound* es un entero no negativo, las condiciones (2) y (3) juntas garantizan que el loop termina.

La relación (1.32) da una idea de cómo entender y desarrollar un loop. Primera, identificar un predicado invariante que es true antes y después de cada iteración. El invariante captura las relaciones *unchanging* entre variables. Segundo, identificar una bounding expression que es no negativa y decrece en cada iteración. Esta expresión captura las relaciones *changing* entre variables, con los cambios resultantes de progresar hacia terminación.

Búsqueda Lineal Revisitada

Consideremos el problema de búsqueda lineal nuevamente. Recordemos que el estado final del programa debe satisfacer:

LS: $a[i] = x \wedge (\forall\ j : 1 \leq j < i: a[j] \neq x)$

Dado que se debe buscar en a, se requiere un loop. Cuando la postcondición de un loop es en forma de conjunción, una técnica para encontrar un invariante es *borrar uno de los conjuntores*. El que se borra generalmente es el que expresa el hecho de que se debe alcanzar terminación; en este caso, a[i] = x. Así, un candidato a invariante para este problema es:

I: $(\forall\ j : 1 \leq j < i: a[j] \neq x)$

La negación del conjuntor borrado luego se usa como guarda del loop. La variable *i* se inicializa de manera que *I* es true antes de la primera iteración. Esto lleva al siguiente esqueleto de programa:

var i:=1
{ *I*: $(\forall\ j : 1 \leq j < i: a[j] \neq x)$ }
do $a[i] \neq x \rightarrow$? **od**

{ $I \wedge a[i]=x$ }

El programa se completa diseñando un cuerpo de loop apropiado. Dentro del cuerpo, los objetivos son reestablecer el invariante y progresar hacia terminación. Aquí, esto se hace incrementando *i*. Esto reestablece el invariante ya que la guarda del loop asegura que *x* aún no se encontró. Incrementar *i* también progresa hacia terminación. Dado que hay a lo sumo *n* elementos para examinar, *n-i* es el número máximo de elementos que quedan examinar. Esta es una bounding expression dado que su rango es no negativo y su valor decrece en cada iteración. Así tenemos el programa final y su proof outline:

var i:=1
{ *I*: $(\forall j : 1 \leq j < i: a[j] \neq x)$ }
do $a[i] \neq x \rightarrow i := i + 1$ **od**
{ $I \wedge a[i]=x$ }

Ordenación

Esta sección desarrolla un algoritmo que ordena un arreglo entero a[1:n} en forma ascendente. Ilustra el desarrollo de loops anidados y también presenta dos técnicas adicionales para encontrar un loop invariante: *reemplazar una constante por una variable* y *combinar pre y post condiciones*.

Para el algoritmo de ordenación, se asume que el estado inicial satisface:

P: $(\forall k: 1 \leq k \leq n: a[k] = A[k])$

donde A es un arreglo de variables lógicas con los valores iniciales de a. El objetivo del algoritmo es establecer:

SORT: $(\forall k: 1 \leq k \leq n: a[k] \leq a[k+1]) \wedge$ a es una permutación de A

Obviamente tenemos que usar un loop para ordenar un arreglo, entonces debemos encontrar un invariante. No se puede usar la técnica anterior. Borrar un conjuntor de SORT no dará un invariante adecuado pues ninguno puede usarse como guarda y ninguno da una guía de cómo el loop establecerá el otro conjuntor.

Dado que un invariante debe ser true inicialmente y luego de cada iteración, con frecuencia es útil examinar las pre y postcondiciones de un loop cuando se desarrolla un invariante. Esto es especialmente verdadero cuando los valores de entrada son modificados. La pregunta es: pueden ponerse las dos aserciones en la misma forma? Aquí, P es un caso especial del segundo conjuntor de SORT dado que el valor inicial de *a* es exactamente A; así *a* es trivialmente una permutación de A. Además, el estado inicial (en el cual el arreglo no está ordenado) es un caso degenerado del primer conjuntor de SORT si *a* no tiene elementos. Así, si cualquiera de las constantes en SORT (1 o n) es reemplazada por una variable cuyo valor inicial convierte en vacío el rango de *k*, tanto P como la versión modificada de SORT tendrán la misma forma. Reemplazando *n* por la variable *i* nos da el predicado:

I: $(\forall k: 1 \leq k < i: a[k] \leq a[k+1]) \wedge$ a es una permutación de A

Esto servirá como un invariante útil si *i* es inicializada a 1: es verdadero inicialmente, será verdadero luego de cada iteración si un elemento más de *a* es puesto en su lugar correcto, llevará a terminación si *i* es incrementada en cada iteración (con bounding expression *n-i*), y sugiere una guarda de loop fácilmente computada de i<n. Gráficamente, el invariante es:

a[1] ... ordenado ... a[i-1] a[i] ... no ordenado ... a[n]

Un esqueleto del algoritmo de ordenación con este invariante es:

{ I }
do $i<n \rightarrow$ poner valor correcto en a[i]; i:=i+1 **od**
{ $I \wedge i \geq n$ } { SORT }

Al comienzo de cada iteración, a[1:i-1] está ordenado; al final, queremos a[1:i] ordenado. Hay dos estrategias para hacer esto. Una es examinar todos los elementos en a[1:n], seleccionar el más chico, e intercambiarlo con a[i]. Esta aproximación se llama *selection sort*. Una segunda estrategia es mover el valor en a[i] al lugar apropiado en a[1:i], corriendo los otros valores como sea necesario para hacer lugar. Esto es llamado *insertion sort*.

Desarrollaremos un algoritmo para insertion sort ya que es una estrategia un poco mejor, especialmente si a está inicialmente cerca del ordenado. Al comienzo de un paso de inserción, sabemos del invariante *I* que a[1:i-1] está ordenado. Necesitamos insertar a[i] en el lugar adecuado para terminar con a[1:i] ordenado. Una manera simple de hacer esto es primero comparar a[i] con a[i-1]. Si están en orden correcto, ya está. Si no, los intercambiamos, y repetimos el proceso comparando a[i-1] (el antiguo a[i]) y a[i-2]. Seguimos hasta que el valor que inicialmente estaba en a[i] ha llegado a la posición correcta.

Nuevamente necesitamos un invariante. Sea *j* el índice del nuevo valor que estamos insertando; inicialmente *j* es igual a *i*. Al final del loop de inserción, queremos que el siguiente predicado sea true:

$$Q: (\forall k: 1 \le k < j-1: a[k] \le a[k+1]) \wedge$$
$$(\forall k: j \le k < i: a[k] \le a[k+1]) \wedge$$
$$(j = 1 \vee a[j-1] \le a[j] \wedge a \text{ es una permutación de } A$$

El primer conjuntor dice que a[1:j-1] está ordenado; el segundo dice que a[j:i] está ordenado; el tercero que a[j] está en el lugar correcto (el cual podría ser a[1]). Nuevamente podemos usar la técnica de borrar un conjuntor para tener un invariante apropiado. Borramos el tercer conjuntor ya que es el único que podría ser falso al comienzo del loop de inserción; en particular, es lo que el loop debe hacer verdadero. Esto da el invariante:

$$II: (\forall k: 1 \le k < j-1: a[k] \le a[k+1]) \wedge$$
$$(\forall k: j \le k < i: a[k] \le a[k+1]) \wedge a \text{ es una permutación de } A$$

Usamos la negación del tercer conjuntor en la guarda del loop. Para mantener invariante *II*, el cuerpo del loop interno intercambia a[j] y a[j-1] y luego decrementa *j*. La bounding expression es *i-j*. El programa completo sería:

```
{ P: (∀ k: 1 ≤ k ≤ n: a[k] = A[k]) }

var i := 1 : int

{ I: (∀ k: 1 ≤ k < i: a[k] ≤ a[k+1]) ∧ a es una permutación de A }
do i < n → j := i
            { II: (∀ k: 1 ≤ k < j-1: a[k] ≤ a[k+1]) ∧
              (∀ k: j ≤ k < i: a[k] ≤ a[k+1]) ∧
              a es una permutación de A }
            do j > 1 and a[j-1] > a[j] →
                  a[j-1] :=: a[j]; j := j - 1
            od
            { II ∧ (j = 1 ∨ a[j-1] ≤ a[j]) }
            i := i + 1
            { I }
od
{ I ∧ i ≥ n }
{ SORT: (∀ k: 1 ≤ k < n: a[k] ≤ a[k+1]) ∧ a es una permutación de A }
```

Concurrencia y Sincronización

Recordemos que un programa concurrente especifica dos o más procesos cooperantes. Cada proceso ejecuta un programa secuencial y es escrito usando la notación introducida en el capítulo anterior. Los procesos interactúan comunicándose, lo cual lleva a la necesidad de sincronización.

Este capítulo introduce notaciones de programación para concurrencia y sincronización. Por ahora, los procesos interactúan leyendo y escribiendo variables compartidas. Este capítulo también examina los conceptos semánticos fundamentales de la programación concurrente y extiende la lógica de programación PL para incorporar estos conceptos.

El problema clave que puede darse en los programas concurrentes es la interferencia, la cual resulta cuando un proceso toma una acción que invalida las suposiciones hechas por otro proceso.

ESPECIFICACION DE LA EJECUCION CONCURRENTE

Cuando se ejecuta un programa secuencial, hay un único *thread* de control: el contador de programa comienza en la primera acción atómica del proceso y se mueve a través del proceso a medida que las acciones atómicas son ejecutadas. La ejecución de un programa concurrente resulta en múltiples thread de control, uno por cada proceso concurrente.

En nuestra notación, especificaremos la ejecución concurrente con la sentencia **co** (con frecuencia llamada sentencia **cobegin**). Sea S_1 un programa secuencial: una secuencia de sentencias secuenciales y declaraciones opcionales de variables locales. La siguiente sentencia ejecuta las S_i concurrentemente:

(2.1) **co** S_1 // ..... // S_n **oc**

El efecto es equivalente a algún interleaving de las acciones atómicas de las S_i. La ejecución del **co** termina cuando todas las S_i teminaron.

Como ejemplo, consideremos el siguiente fragmento de programa:

```
(2.2)   x := 0; y := 0
        co x := x + 1 // y := y + 1 oc
        z := x + y
```

Esto asigna secuencialmente 0 a x e y, luego incrementa x e y concurrentemente (o en algún orden arbitrario), y finalmente asigna la suma de x e y a z. Las variables x e y son globales a los procesos en la sentencia **co**; son declaradas en el proceso que engloba y son heredadas por los procesos en el **co** siguiendo las reglas de alcance normales de los lenguajes estructurados. Un proceso también puede declarar variables *locales* cuyo alcance se limita a ese proceso.

Con frecuencia un programa concurrente contiene un número de procesos que realizan la misma computación sobre diferentes elementos de un arreglo. Especificaremos esto usando cuantificadores en las sentencias **co**, con los cuantificadores idénticos en forma a los de la sentencia **fa**. Por ejemplo, la siguiente sentencia especifica *n* procesos que en paralelo inicializan todos los elementos de a[1:n] en 0:

co i := 1 **to** n $\rightarrow$ a[i] := 0 **oc**

Cada proceso tiene una copia local de i, la cual es una constante entera declarada implícitamente; el valor de i está entre 1 y n y es distinta en cada proceso. Así, cada proceso tiene una identidad única.

Como segundo ejemplo, el siguiente programa multiplica las matrices a y b, ambas nxn, en paralelo, poniendo el resultado en la matriz c:

```
(2.3)   co i := 1 to n, j := 1 to n →
            var sum : real := 0
            fa k := 1 to n → sum := sum + a[i, k] * b[k, j] af
            c[i, j] := sum
        oc
```

Cada una de los n^2 procesos tiene constantes locales *i* y *j* y variables locales *sum* y *k*. El proceso (*i*, *j*) computa el producto interno de la fila *i* de *a* y la columna *j* de *b* y almacena el resultado en *c*[*i*, *j*].

Cuando hay muchos procesos (o los procesos son largos) puede ser inconveniente especificar la ejecución concurrente usando solo la sentencia **co**. Esto es porque el lector puede perder la pista del contexto. En consecuencia, emplearemos una notación alternativa como se muestra en el siguiente programa, que busca el valor del elemento más grande de a y en paralelo asigna a cada b[i] la suma de los elementos de a[1:i]:

```
var a[1:n], b[1:n] : int
Largest :: var max : int
           max := a[1]
           fa j := 2 to n st max < a[j] → max := a[j] af
Sum [i:1..n] :: b[i] := a[1]
                fa j := 2 to i → b[i] := b[i] + a[j] af
```

Largest es el nombre de un solo proceso. *Sum* es un arreglo de *n* procesos; cada elemento *Sum*[i] tiene un valor diferente para *i*, el cual es una variable entera local declarada implícitamente. Los procesos en este ejemplo son los mismos que si hubiéramos hecho:

co cuerpo de *Largest* // i := 1 **to** n → cuerpo de *Sum* **oc**

Dado que la notación del ejemplo es solo una abreviación de la sentencia **co**, la semántica de la ejecución concurrente depende solo de la semántica de **co**.

ACCIONES ATOMICAS Y SINCRONIZACION

Como ya dijimos, podemos ver la ejecución de un programa concurrente como un interleaving de las acciones atómicas ejecutadas por procesos individuales. Cuando los procesos interactúan, no todos los interleavings son aceptables. El rol de la sincronización es prevenir los interleavings indeseables. Esto se hace combinando acciones atómicas fine-grained en acciones (compuestas) coarse grained, o demorando la ejecución de un proceso hasta que el estado de programa satisfaga algún predicado. La primera forma de sincronización se llama *exclusión mutua;* la segunda, *sincronización por condición*.

Atomicidad Fine-Grained

Recordemos que una acción atómica hace una transformación de estado indivisible. Esto significa que cualquier estado intermedio que podría existir en la implementación de la acción no debe ser visible para los otros procesos. Una acción atómica *fine-grained* es implementada directamente por el hardware sobre el que ejecuta el programa concurrente.

Para que el axioma de asignación sea *sound*, cada sentencia de asignación debe ser ejecutada como una acción atómica. En un programa secuencial, las asignaciones aparecen como atómicas ya que ningún estado intermedio es visible al programa. Sin embargo, esto en general no ocurre en los programas concurrentes, ya que una asignación con frecuencia es implementada por una secuencia de instrucciones de máquina fine-grained. Por ejemplo, consideremos el siguiente programa, y asumamos que las acciones atómicas *fine-grained* están leyendo y escribiendo las variables:

```
y := 0;  x := 0
co x := y + z // y := 1; z := 2 oc
```

Si x := y + z es implementada cargando un registro con y, luego sumándole z, el valor final de x podría ser 0, 1, 2 o 3. Esto es porque podríamos ver los valores iniciales para *y* y *z*, sus valores finales, o alguna combinación, dependiendo de cuan rápido se ejecuta el segundo proceso. Otra particularidad del programa anterior es que el valor final de x podría ser 2, aunque y+z no es 2 en ningún estado de programa.

Asumiremos que las máquinas que ejecutan los programas tienen las siguientes características:

* Los valores de los tipos básicos y enumerativos (por ej, **int**) son almacenados en elementos de memoria que son leídos y escritos como acciones atómicas. (Algunas máquinas tienen otras instrucciones indivisibles, tales como incrementar una posición de memoria o mover los contenidos de una posición a otra).

* Los valores son manipulados cargándolos en registros, operando sobre ellos, y luego almacenando los resultados en memoria.

* Cada proceso tiene su propio conjunto de registros. Esto se realiza teniendo distintos conjuntos de registros o salvando y recuperando los valores de los registros cada vez que se ejecuta un proceso diferente. (Esto se llama *context switch*).

* Todo resultado intermedio de evaluar una expresión compleja se almacena en registros o en memoria privada del proceso ejecutante (por ejemplo, en una pila privada).

Con este modelo de máquina, si una expresión *e* en un proceso no referencia una variable alterada por otro proceso, la evaluación de expresión será atómica, aún si requiere ejecutar muchas acciones atómicas fine-grained. Esto es porque ninguno de los valores de los que depende *e* podría cambiar mientras *e* está siendo evaluada y porque cada proceso tiene su propio conjunto de registros y su propia área de almacenamiento temporario. De manera similar, si una asignación x:=e en un proceso no referencia ninguna variable alterada por otro proceso, la ejecución de la asignación será atómica. El programa en (2.2) cumple este requerimiento, y por lo tanto las asignaciones concurrentes son atómicas.

Desafortunadamente, los programas concurrentes más interesantes no cumplen el requerimiento de ser disjuntos. Sin embargo, con frecuencia se cumple un requerimiento más débil. Definimos una variable *simple* como una variable escalar, elemento de arreglo o campo de registro que es almacenada en una posición de memoria única. Entonces si una expresión o asignación satisface la siguiente propiedad, la evaluación aún será atómica:

(2.4) **Propiedad de "a lo sumo una vez"**. Una expresión *e* satisface la propiedad de "a lo sumo una vez" si se refiere a lo sumo a una variable simple *y* que podría ser cambiada por otro proceso mientras *e* está siendo evaluada, y se refiere a *y* a lo sumo una vez. Una sentencia de asignación x:=e satisface esta propiedad si *e* satisface la propiedad y *x* no es leída por otro proceso, o si *x* es una variable simple y *e* no se refiere a ninguna variable que podría ser cambiada por otro proceso.

Esta propiedad asegura atomicidad ya que la variable compartida, si la hay, será leída o escrita solo una vez como una acción atómica fine-grained.

Para que una sentencia de asignación satisfaga (2.4), *e* se puede referir a una variable alterada por otro proceso si x no es leída por otro proceso (es decir, *x* es una variable local). Alternativamente, *x* puede ser leída por otro proceso si *e* no referencia ninguna variable alterada por otro proceso. Sin embargo, ninguna asignación en lo siguiente satisface (2.4):

co x := y + 1 // y := x + 1 **oc**

En realidad, si x e y son inicialmente 0, sus valores finales podrían ser ambos 1 (Esto resulta si el proceso lee x e y antes de cualquier asignación a ellas). Sin embargo, dado que cada asignación se refiere sólo una vez a solo una variable alterada por otro proceso, los valores finales serán algunos de los que realmente existieron en algún estado. Esto contrasta con el ejemplo anterior en el cual y+z se refería a dos variables alteradas por otro proceso.

Especificación de la Sincronización

Si una expresión o sentencia de asignación no satisface la propiedad de a lo sumo una vez, con frecuencia necesitamos tener que ejecutarla atómicamente. Más generalmente, necesitamos ejecutar secuencias de sentencias como una única acción atómica. En ambos casos, necesitamos usar un mecanismo de sincronización para construir una acción atómica *coarse grained*, la cual es una secuencia de acciones atómicas fine grained que aparecen como indivisibles.

Como ejemplo concreto, supongamos una BD que contiene dos valores x e y, y que en todo momento x e y deben ser lo mismo en el sentido de que ningún proceso que examina la BD puede ver un estado en el cual x e y difieren. Luego, si un proceso altera x, debe también alterar y como parte de la misma acción atómica.

Como segundo ejemplo, supongamos que un proceso inserta elementos en una cola representada por una lista enlazada. Otro proceso remueve elementos de la lista, asumiendo que hay elementos. Dos variables apuntan a la cabeza y la cola de la lista. Insertar y remover elementos requiere manipular dos valores; por ejemplo, para insertar un elemento, tenemos que cambiar el enlace del que era anteriormente último elemento para que apunte al nuevo elemento, y tenemos que cambiar la variable que apunta al último para que apunte al nuevo elemento. Si la lista contiene solo un elemento, la inserción y remoción simultánea puede conflictuar, dejando la lista en un estado inestable. Así, insertar y remover deben ser acciones atómicas. Además, si la lista está vacía, necesitamos demorar la ejecución de remover hasta que se inserte un elemento.

Especificaremos acciones atómicas por medio de corchetes angulares ⟨ y ⟩. Por ejemplo, ⟨e⟩ indica que la expresión *e* debe ser evaluada atómicamente.

Especificaremos sincronización por medio de la sentencia **await**:

⟨ **await** B → S ⟩

La expresión booleana B especifica una condición de demora; S es una secuencia de sentencias que se garantiza que termina. Una sentencia **await** se encierra en corchetes angulares para indicar que es ejecutada como una acción atómica. En particular, se garantiza que B es true cuando comienza la ejecución de S, y ningún estado interno de S es visible para los otros procesos. Por ejemplo:

(2.5) ⟨ **await** s > 0 → s := s - 1⟩

se demora hasta que s es positiva, y luego la decrementa. El valor de s se garantiza que es positivo antes de ser decrementado.

La sentencia **await** es muy poderosa ya que puede ser usada para especificar acciones atómicas arbitrarias coarse grained. Esto la hace conveniente para expresar sincronización (más adelante la usaremos para desarrollar soluciones iniciales a problemas de sincronización). Este poder expresivo también hace a **await** muy costosa de implementar en su forma más general. Sin embargo, hay casos en que puede ser implementada eficientemente. Por ejemplo, (2.5) es un ejemplo de la operación **P** sobre el semáforo s.

La forma general de la sentencia **await** especifica tanto exclusión mutua como sincronización por condición. Para especificar solo exclusión mutua, abreviaremos una sentencia **await** como sigue:

⟨S⟩

Por ejemplo, lo siguiente incrementa x e y atómicamente:

⟨ x := x + 1 ; y := y + 1 ⟩

El estado interno en el cual x fue incrementada pero y no es invisible a los otros procesos que referencian x o y. Si S es una única sentencia de asignación y cumple los requerimientos de la propiedad (2.4) (o si S es implementada por una única instrucción de máquina) entonces S será ejecutada atómicamente; así, ⟨S⟩ tiene el mismo efecto que S.

Para especificar solo sincronización por condición, abreviaremos una sentencia **await** como:

⟨ **await** B ⟩

Por ejemplo, lo siguiente demora el proceso ejecutante hasta que count sea mayor que 0:

⟨ **await** count > 0 ⟩

Si B cumple los requerimientos de la propiedad (2.4), como en este ejemplo, entonces ⟨**await** B⟩ puede ser implementado como:

do not B → **skip od**

Una acción atómica *incondicional* es una que no contiene una condición de demora B. Tal acción puede ejecutarse inmediatamente, sujeta por supuesto al requerimiento de que se ejecute atómicamente. Las acciones fine grained (implementadas por hardware), las expresiones en corchetes angulares, y las sentencias **await** en la cual la guarda es la constante true o se omite son todas acciones atómicas incondicionales.

Una acción atómica *condicional* es una sentencia **await** con una guarda B. Tal acción no puede ejecutarse hasta que B sea true. Si B es false, solo puede volverse true como resultado de acciones tomadas por otros procesos. Así, un proceso que espera ejecutar una acción atómica condicional podría esperar un tiempo arbitrariamente largo.

SEMANTICA DE LA EJECUCION CONCURRENTE

La sentencia **co** en (2.2) incrementa tanto x como y. Así, una lógica de programación para concurrencia debería permitir probar lo siguiente:

(2.6) { x = 0 ∧ y = 0 }
co x := x + 1 // y := y + 1 **oc**
{ x = 1 ∧ y = 1 }

Esto requiere una regla de inferencia para la sentencia **co**. Dado que la ejecución del **co** resulta en la ejecución de cada proceso, el efecto del **co** es la conjunción de los efectos de los procesos constituyentes. Así, la regla de inferencia para el **co** se basa en combinar triplas que capturan el efecto de ejecutar cada proceso.

Cada proceso comprende sentencias secuenciales, más posiblemente sentencias de sincronización. Los axiomas y reglas de prueba para sentencias secuenciales son las que ya vimos. Con respecto a la corrección parcial, una sentencia **await** es como una sentencia **if** para la cual la guarda B es true cuando comienza la ejecución de S. Por lo tanto, la regla de inferencia para **await** es similar a la Regla de Alternativa:

(2.7) **Regla de Sincronización**: $\dfrac{\{ P \wedge B \}\ S'\ \{ Q \}}{\{ P \}\ \langle\ \textbf{await}\ B \rightarrow S\ \rangle\ \{ Q \}}$

Las dos formas especiales de la sentencia **await** son casos especiales de esta regla. Para $\langle\ S\ \rangle$, B es true, y por lo tanto la hipótesis se simplifica a { P } S { Q }. Para $\langle$ **await** B $\rangle$, S es **skip**, entonces P $\wedge$ B debe implicar la verdad de Q.

Como ejemplo del uso de (2.7), por el axioma de asignación el siguiente es un teorema:

$\{ s > 0 \}\ s := s - 1\ \{ s \geq 0 \}$

Luego, por la regla de sincronización

$\{ s \geq 0 \}\ \langle$ **await** $s > 0 \rightarrow s := s - 1\ \rangle\ \{ s \geq 0 \}$

es un teorema con P y Q ambos siendo $s \geq 0$. El hecho de que las sentencias **await** se ejecuten atómicamente afecta la interacción entre procesos, como se discute abajo.

Supongamos que para cada proceso S_i en una sentencia **co** de la forma (2.1),

$\{ P_i \}\ S_i\ \{ Q_i \}$

es un teorema de PL. De acuerdo a la Interpretación de Triplas (1.5), esto significa que, si S_i se inicia en un estado que satisface P_i y S_i termina, entonces el estado va a satisfacer Q. Para que esta interpretación se mantenga cuando los procesos son ejecutados concurrentemente, los procesos deben ser iniciados en un estado que satisfaga la conjunción de las P_i. Si todos los procesos terminan, el estado final va a satisfacer la conjunción de las Q_i. Así, uno esperaría poder concluir que lo siguiente es un teorema:

$\{ P_1 \wedge .... \wedge P_n \}$ **co** S_1 // .... // S_n **oc** $\{ Q_1 \wedge .... \wedge Q_n \}$

Para el programa (2.6), tal conclusión sería sound. En particular, a partir de las triplas válidas

$\{ x = 0 \}\ x := x + 1\ \{ x = 1 \}$
$\{ y = 0 \}\ y := y + 1\ \{ y = 1 \}$

la siguiente es una conclusión sound:

$\{ x = 0 \wedge y = 0 \}$ **co** x := x + 1 // y := y + 1 **oc** $\{ x = 1 \wedge y = 1 \}$

Pero qué sucede con el siguiente programa?

co $\langle$ x := x + 1 $\rangle$ // $\langle$ x := x + 1 $\rangle$ **oc**

Las sentencias de asignación son atómicas, luego si x es inicialmente 0, su valor final es 2. Pero cómo podemos probar esto? Aunque lo siguiente es un teorema aisladamente

$\{ x = 0 \}\ \langle\ x := x + 1\ \rangle\ \{ x = 1 \}$

no es generalmente válido cuando otro proceso ejecuta concurrentemente y altera la variable compartida x. El problema es que un proceso podría *interferir* con una aserción en el otro; es decir, podría convertir en falsa la aserción.

Asumimos que cada expresión o sentencia de asignación se ejecuta atómicamente, o porque cumple los requerimientos de la propiedad de a lo sumo una vez o porque está entre corchetes angulares. Una acción atómica en un proceso es *elegible* si es la próxima acción atómica que el proceso ejecutará. Cuando se ejecuta una acción elegible T, su precondición pre(T) debe ser true. Por ejemplo, si la próxima acción elegible es evaluar las guardas de una sentencia **if**, entonces la precondición de la sentencia **if** debe ser true. Dado que las acciones elegibles en distintos procesos se ejecutan en cualquier orden, pre(T) no debe volverse falsa si alguna otra acción elegible se ejecuta antes que T. Llamamos al predicado pre(T) *aserción crítica* dado que es imperativo que sea true cuando se ejecuta T. Más precisamente, lo siguiente define el conjunto de aserciones críticas en una prueba.

(2.8) **Aserciones Críticas**. Dada una prueba de que { P } S { Q } es un teorema, las aserciones críticas en la prueba son: (a) Q, y (b) para cada sentencia T dentro de S que no está dentro de una sentencia **await**, el predicado weakest pre(T) tal que { pre(T) } T { post(T) } es un teorema dentro de la prueba.

En el caso (b), solo las sentencias que no están dentro de los **await** necesitan ser consideradas dado que los estados intermedios dentro del **await** no son visibles a los otros procesos. Además, si hay más de una tripla válida acerca de una sentencia dada T (debido al uso de la regla de consecuencia) sólo el predicado weakest pre(T) que es precondición de T es una aserción crítica. Esto es porque no se necesitan suposiciones más fuertes para construir la prueba entera.

Recordemos que una proof outline completa contiene una aserción antes y después de cada sentencia. Codifica una prueba formal, presentándola de una manera que la hace más fácil de entender. También codifica las aserciones críticas, asumiendo que son las weakest requeridas. En particular, las aserciones críticas de un proceso son la postcondición y las precondiciones de cada sentencia que no está dentro de un **await**. Por ejemplo, en

$$\{ P_e^x \} \langle x := e \rangle \{ P \}$$

P_e^x es una aserción crítica ya que debe ser true cuando la sentencia de asignación se ejecuta; P también es una aserción crítica si la asignación es la última sentencia en un proceso. Como segundo ejemplo, en

$$\{ P \} \langle S1; \{ Q \} S2 \rangle \{ R \}$$

P es una aserción crítica, pero Q no, ya que el estado siguiente a la ejecución de S1 no es visible a los otros procesos.

Para que la prueba de un proceso se mantenga válida de acuerdo a la Interpretación para Triplas (1.5), las aserciones criticas no deben ser interferidas por acciones atómicas de otros procesos ejecutando concurrentemente. Esto se llama *libertad de interferencia*, lo cual definiremos más formalmente.

Una *acción de asignación* es una acción atómica que contiene una o más sentencias de asignación. Sea C una aserción crítica en la prueba de un proceso. Entonces la única manera en la cual C podría ser interferida es si otro proceso ejecuta una acción de asignación *a* y *a* cambia el estado de manera que C sea falsa. La interferencia no ocurrirá si se mantienen las siguientes condiciones:

(2.9) **No Interferencia**. Si es necesario, renombrar las variables locales en C para que sus nombres sean distintos de los de las variables locales en *a* y *pre(a)*. Entonces la acción de asignación *a* no interfiere con la aserción crítica C si lo siguiente es un teorema en la Lógica de Programación:

NI(*a*, C) : { C $\wedge$ *pre(a)* } *a* { C }

En resumen, C es invariante con respecto a la ejecución de la acción de asignación *a*. La precondición de *a* se incluye en (2.9) ya que *a* puede ser ejecutada solo si el proceso está en un estado que satisface *pre(a)*.

Un conjunto de procesos está libre de interferencia si ninguna acción de asignación en un proceso interfiere con ninguna aserción crítica en otro. Si esto es verdad, entonces las pruebas de los procesos individuales se mantienen verdaderas en presencia de la ejecución concurrente.

(2.10) **Libertad de Interferencia**. Los teoremas $\{ P_i \}\ S_i\ \{ Q_i \}$, $1 \leq i \leq n$, son libres de interferencia si:

Para todas las acciones de asignación a en la prueba de S_i,
Para todas las aserciones críticas C en la prueba de S_j, $i \neq j$,
NI(*a*, C) es un teorema.

Si las pruebas de los n procesos son libres de interferencia, entonces las pruebas pueden ser combinadas cuando los procesos se ejecutan concurrentemente. En particular, si los procesos comienzan la ejecución en un estado que satisface todas sus precondiciones, y si todos los procesos terminan, entonces cuando los procesos terminan el estado va a satisfacer la conjunción de sus postcondiciones. Esto lleva a la siguiente regla de inferencia para **co**.

(2.11) **Regla de Concurrencia**:

$\{ P_i \}\ S_i\ \{ Q_i \}$ son teoremas libres de interferencia, $1 \leq i \leq n$

$\{ P_1 \wedge .... \wedge P_n \}$ **co** S_1 // .... // S_n **oc** $\{ Q_1 \wedge .... \wedge Q_n \}$

TECNICAS PARA EVITAR INTERFERENCIA

Para aplicar la regla de concurrencia (2.11), primero necesitamos construir pruebas de los procesos individuales. Dado que los procesos son programas secuenciales, las pruebas se desarrollan usando las técnicas ya descriptas. Luego tenemos que mostrar que las pruebas están libres de interferencia. Este es el nuevo requerimiento introducido por la ejecución concurrente.

Recordemos que el número de historias distintas de un programa concurrente es exponencial con respecto al número de acciones atómicas que se ejecutan. Por contraste, el número de maneras en que los procesos pueden interferir depende solo del número de acciones atómicas distintas; este número no depende de cuantas acciones son ejecutadas realmente. Por ejemplo, si hay *n* procesos y cada uno contiene *a* acciones de asignación y *c* aserciones críticas, entonces en el peor caso tenemos que probar n * (n-1) * a * c teoremas de no interferencia. Aunque este número es mucho menor que el número de historias, igual es bastante grande. Sin embargo, muchos de estos teoremas serán los mismos, ya que con frecuencia los procesos son idénticos sintácticamente o al menos simétricos. Además, hay maneras de evitar completamente la interferencia.

Presentaremos 4 técnicas para evitar interferencia: variables disjuntas, aserciones weakened, invariantes globales y sincronización. Todas involucran aserciones y acciones de asignación puestas en una forma que asegure que las fórmulas de no interferencia (2.9) son verdaderas.

Variables Disjuntas

El *write set* de un proceso es el conjunto de variables que asigna. El *reference set* de un proceso es el conjunto de variables referenciadas en las aserciones en una prueba de ese proceso. (Con frecuencia es el mismo que el conjunto de variables referenciadas en sentencias del proceso, pero podría no serlo; con respecto a interferencia, las variables críticas son las de las aserciones).

Si el write set de un proceso es disjunto del reference set de otro, y viceversa, entonces los procesos no pueden interferir. Esto es porque el axioma de asignación (1.7) emplea sustitución textual, la cual no tiene efecto sobre un predicado que no contiene una referencia al destino de la asignación (Las variables locales con igual nombre en distintos procesos pueden ser renombradas para aplicar el axioma de asignación).

Como ejemplo, consideremos nuevamente el siguiente programa:

co x := x + 1 // y := y + 1 **oc**

Si x e y son inicialmente 0, entonces a partir del axioma de asignación, los siguientes son teoremas:

{ x = 0 } x := x + 1 { x = 1 }
{ y = 0 } y := y + 1 { y = 1 }

Cada proceso contiene una sentencia de asignación y dos aserciones; por lo tanto hay que probar 4 teoremas de no interferencia. Por ejemplo, para mostrar que x = 0 no es interferida con la asignación a y,

NI(y:=y+1, x=0): { x = 0 $\wedge$ y = 0 } y := y + 1 { x = 0 }

debe ser un teorema. Esto es porque

$(x=0)^{y}_{y+1} = (x=0)$

y (x=0 $\wedge$ y=0) $\Rightarrow$ x=0. Las otras tres pruebas de no interferencia son similares dado que los conjuntos write y reference son disjuntos. Así, podemos aplicar la Regla de Concurrencia (2.11) para concluir que el siguiente es un teorema:

{ x=0 $\wedge$ y=0 } **co** x := x + 1 // y := y + 1 **oc** { x=1 $\wedge$ y=1 }

Técnicamente, si una aserción en un proceso referencia una variable arreglo, el arreglo entero es una parte del reference set del proceso. Esto es porque el axioma de asignación trata los arreglos como funciones. Sin embargo, asignar a un elemento del arreglo no afecta el valor de ningún otro. Por lo tanto si los elementos del arreglo reales en el write set de un proceso difieren del los elementos del arreglo reales en el reference set del otro (y los conjuntos son en otro caso disjuntos) el primer proceso no interfiere con el segundo. Por ejemplo, en

co i := 1 **to** n $\rightarrow$ a[i] := i **oc**

la siguiente tripla es válida para cada proceso:

{true} a[i] := i { a[i]=i }

Dado que el valor de i es distinto en cada proceso, las pruebas son libres de interferencia. Análogamente, los procesos producto interno en (2.3) y los procesos *Sum* no interferirían uno con otro si sus pruebas no referencian elementos de arreglo asignados por otros procesos. En casos simples como estos, es fácil ver que los índices de los arreglos son diferentes; en general, esto puede ser difícil de verificar, si no imposible.

Los write/reference sets disjuntos proveen la base para muchos algoritmos paralelos, especialmente los del tipo de (2.3) que manipulan matrices. Como otro ejemplo, las diferentes ramas del árbol de posibles movimientos en un programa de juegos pueden ser buscadas en paralelo. O múltiples transacciones pueden examinar una BD en paralelo, o pueden actualizar distintas relaciones.

Aserciones Weakened

Aún cuando los write y reference sets de los procesos se overlapen, a veces podemos evitar interferencia debilitando aserciones para tomar en cuenta los efectos de la ejecución concurrente. Por ejemplo, consideremos lo siguiente:

(2.12) **co** P1: ⟨ x := x + 1 ⟩ // P2: ⟨ x := x + 2 ⟩ **oc**

Si x es inicialmente 0, las siguientes triplas son ambas válidas aisladamente:

{ x = 0 } x := x + 1 { x = 1 }

{ x = 0 } x := x + 2 { x = 2 }

Sin embargo, cada asignación interfiere con ambas aserciones de la otra tripla. Además, la conjunción de las postcondiciones no da el resultado correcto de x=3.

Si el proceso P1 se ejecuta antes que P2, entonces el estado va a satisfacer x=1 cuando P2 comienza la ejecución. Si debilitamos la precondición de P2 para tomar en cuenta esta posibilidad, resulta la siguiente tripla:

(2.13) { x = 0 $\vee$ x = 1 } x := x + 2 { x = 2 $\vee$ x = 3 }

Análogamente, si P2 se ejecuta antes que P1, el estado va a satisfacer x=2 cuando P1 comienza su ejecución. Así, también necesitamos debilitar la precondición de P1:

(2.14) { x = 0 $\vee$ x = 2 } x := x + 1 { x = 1 $\vee$ x = 3 }

Las pruebas no interfieren. Por ejemplo, la precondición y asignación en (2.14) no interfieren con la precondición en (2.13). Aplicando la definición de no interferencia (2.9):

(2.15) { (x = 0 $\vee$ x = 1) $\wedge$ (x = 0 $\vee$ x = 2) } x := x + 1 { x = 0 $\vee$ x = 1 }

Sustituyendo x+1 por x en la postcondición de (2.15) se tiene (x = -1 $\vee$ x = 0). La precondición en (2.15) se simplifica a x=0, lo cual implica (x = -1 $\vee$ x = 0). Aplicando la regla de consecuencia para librarnos de x = -1, (2.15) es un teorema.

Los otros tres chequeos de no interferencia son similares. Por lo tanto, podemos aplicar la regla de concurrencia (2.11) a (2.13) y (2.14), obteniendo el teorema:

{ (x = 0 $\vee$ x = 1) $\wedge$ (x = 0 $\vee$ x = 2) }
co P1: $\langle$ x := x + 1 $\rangle$ // P2: $\langle$ x := x + 2 $\rangle$ **oc**
{ (x = 2 $\vee$ x = 3) $\wedge$ (x = 1 $\vee$ x = 3) }

La precondición se simplifica a x = 0, y la postcondición se simplifica a x = 3, como queríamos.

Aunque el programa anterior es simple, ilustra un principio importante: al desarrollar una prueba de un proceso que referencia variables alteradas por otro proceso, tomar en cuenta los efectos de los otros procesos. El ejemplo también ilustra una manera de hacerlo: hacer una aserción más débil acerca de las variables compartidas que la que podría hacerse si un proceso se ejecutara aisladamente. Siempre podemos debilitar lo suficiente las aserciones para evitar interferencia (por ej, cualquier estado satisface *true*) pero entonces el resultado deseado probablemente no pueda ser probado.

Invariantes Globales

Otra técnica para evitar interferencia es emplear un invariante global para capturar la relación entre variables compartidas. Supongamos que *I* es un predicado que referencia variables globales. Entonces *I* es un *invariante global* con respecto a un conjunto de procesos si (1) *I* es true cuando los procesos comienzan la ejecución, y (2) *I* es invariante con respecto a la ejecución de cualquier acción de asignación (sentencias de asignación o **await** que contienen una asignación). La condición 1 se satisface si *I* es true en el estado inicial de cada proceso. La condición 2 se satisface si para cada acción de asignación *a* con precondición *pre(a)*,

{ *I* $\wedge$ *pre(a)* } *a* { *I* }

es un teorema; es decir, *a* no interfiere con *I*. Recordemos que *pre(a)* se incluye dado que *a* solo puede ser ejecutada en un estado en el cual *pre(a)* es true.

Supongamos que *I* es un invariante global. También supongamos que toda aserción crítica C en la prueba de un proceso P_j puede ponerse en la forma:

(2.16) C: I $\wedge$ L

donde L es un predicado sobre variables privadas. En particular, todas las variables referenciadas en L o son locales al proceso j o son variables globales que solo asigna j.

Si todas las aserciones pueden ponerse en la forma (2.16), entonces las pruebas de los procesos están libres de interferencia. Esto es porque I es invariante con respecto a cualquier acción de asignación *a* y porque ninguna acción de asignación en un proceso puede interferir con un predicado local L en otro proceso pues los destinos en *a* son distintos de todas la variables de L. Así, el requerimiento de no interferencia (2.19) se cumple para cada par de acciones de asignación y aserciones críticas. Además, solo tenemos que chequear las triplas en cada proceso para verificar que cada aserción critica tiene la forma anterior y que I es un invariante global; no tenemos que considerar las aserciones o sentencias en otros procesos. De hecho, para un arreglo de procesos idénticos, solo tenemos que chequear uno de ellos.

Aún cuando no podemos poner toda aserción crítica en la forma (2.16), a veces podemos usar una combinación de invariantes globales y aserciones debilitadas para evitar interferencia. Ilustramos esto usando el programa de productor/consumidor:

```
var buf : int, p : int, c : int := 0
Productor :: var a[1:n] : int
             do p < n → ⟨ await p = c ⟩
                        buf := a[p+1]
                        p := p + 1
             od
Consumidor :: var b[1:n] : int
              do c < n → ⟨ await p > c ⟩
                         b[c+1] := buf
                         c := c + 1
              od
```

Este programa copia los contenidos del arreglo *Productor a* en el arreglo *Consumidor b* usando un buffer simple compartido *buf*. El *Productor* y el *Consumidor* alternan el acceso a buf. Las variables p y c cuentan el número de items que han sido depositados y buscados, respectivamente. Las sentencias **await** se usan para sincronizar el acceso a buf. Cuando p = c el buffer está vacío, cuando p > c el buffer está lleno.

Supongamos que los contenidos iniciales de a[1:n] son A[1:n], donde A es un arreglo de variables lógicas. El objetivo es probar que, al terminar el programa anterior, los contenidos de b[1:n] son A[1:n]. Esto se puede hacer usando un invariante global como sigue. Dado que los procesos se alternan el acceso a buf, en todo momento p es igual a c o a uno más que c. También, cuando el buffer está lleno (es decir, p=c+1), contiene A[p]. Finalmente, a no es alterada, por lo tanto siempre es igual a A. Así, un candidato a invariante global es:

$$PC: c \leq p \leq c + 1 \ \wedge \ a[1:n] = A[1:n] \ \wedge \ (p = c + 1) \Rightarrow (buf = A[p])$$

Este predicado es true inicialmente ya que p = c = 0. Esto es mantenido por cada sentencia de asignación, como se ve en la proof outline completa:

```
var buf : int, p : int, c : int := 0
{ PC: c ≤ p ≤ c + 1 ∧ a[1:n] = A[1:n] ∧ (p = c + 1) ⇒ ( buf = A[p] ) }
Productor :: var a[1:n] : int
             { IP: PC ∧ p ≤ n }
             do p < n → { PC ∧ p < n }
                        ⟨ await p = c ⟩
                        { PC ∧ p < n ∧ p = c}
                        buf := a[p+1]
                        { PC ∧ p < n ∧ p = c ∧ buf = A[p+1] }
                        p := p + 1
                        { IP }
             od
```

```
                { PC ∧ p = n }
Consumidor :: var b[1:n] : int
                { IC: PC ∧ c ≤ n ∧ b[1:c] = A[1:c] }
                do c < n → { IC ∧ c < n }
                           ⟨ await p > c ⟩
                           { IC ∧ c < n ∧ p > c}
                           b[c+1] := buf
                           { IC ∧ c < n ∧ p > c ∧ b[c+1] = A[c+1] }
                           c := c + 1
                           { IC }
                od
                { IC ∧ c = n + 1 }
```

Esta prueba contiene todas las aserciones críticas. IP es el invariante para el loop productor e IC para el loop consumidor.

Aisladamente, la proof outline para cada proceso del ejemplo anterior se deduce directamente de las sentencias de asignación del proceso. Asumiendo que cada proceso continuamente tiene chance de ejecutar, las sentencias **await** terminan dado que primero es true una guarda, luego la otra, etc. Dado que las sentencias **await** terminan, cada proceso termina luego de *n* iteraciones. Así, como las pruebas son libres de interferencia, podemos usar la regla de concurrencia (2.11) para combinar las postcondiciones de los procesos y concluir que el programa copia los contenidos de a en b.

La mayoría de las aserciones críticas en el ejemplo anterior tienen la forma (2.16): contienen la conjunción del invariante global PC y predicados que referencian variables que no son alteradas por los otros procesos. Por lo tanto las aserciones no pueden ser interferidas. Las únicas aserciones que no siguen esta forma son las dos en *Productor* que afirman que p = c y las dos en *Consumidor* que afirman que p > c. Tenemos que chequear si se interfieren.

Consideremos la siguiente aserción en *Productor*:

A1: { PC ∧ p < n ∧ p = c}

La única variable en A1 que es alterada por *Consumidor* es c. Una sentencia de asignación en *Consumidor* altera c, y esa sentencia tiene precondición:

A2: { IC ∧ c < n ∧ p > c ∧ b[c+1] = A[c+1] }

Aplicando el requerimiento de no interferencia (2.19), tenemos la siguiente obligación de prueba:

NI(c:=c+1,A1): { A1 ∧ A2 } c := c + 1 { A1 }

Dado que p no puede ser a la vez igual y mayor que c, (A1 ∧ A2) es falsa. Por lo tanto, podemos usar la regla de consecuencia para obtener cualquier predicado, en este caso $A1^{c}_{c+1}$. Por el axioma de asignación y la regla de consecuencia, lo anterior es un teorema. Las otras tres pruebas de no interferencia son casi idénticas. Por lo tanto el programa está libre de interferencia.

Lo que ocurre es que las sentencias **await** aseguran que los procesos alternan el acceso al buffer; es decir, p y c alternativamente son iguales y luego difieren en uno. Esto resguarda a las sentencias que incrementan p y c de interferir con aserciones críticas en el otro proceso.

Dado que cada sentencia **await** en el ejemplo cumple los requerimientos de la propiedad de a lo sumo una vez, cada una puede implementarse con un loop **do**. Por ejemplo, la sentencia ⟨**await** p = c ⟩ puede ser implementada por:

do p ≠ c → **skip od**

Cuando la sincronización se implementa de esta manera, un proceso se dice que está en *busy waiting* o *spinning*, ya que está ocupado haciendo nada más que un chequeo de la guarda.

Sincronización

Como ya describimos, podemos ignorar una sentencia de asignación que está dentro de corchetes angulares cuando consideramos obligaciones de no interferencia. Dado que una acción atómica aparece hacia los otros procesos como una unidad indivisible, alcanza con establecer que la acción entera no causa interferencia. Por ejemplo, dado

⟨ x := x + 1; y := y + 1 ⟩

ninguna asignación por si misma puede causar interferencia; solo podría el par de asignaciones.

Además, los estados internos de los segmentos de programa dentro de los corchetes no son visibles. Por lo tanto, ninguna aserción respecto un estado interno puede ser interferida por otro proceso. Por ejemplo, la siguiente aserción del medio no es una aserción crítica:

{ x=0 ∧ y=0 } ⟨ x := x + 1 { x=1 ∧ y=0 } y := y + 1⟩ { x=1 ∧ y=1 }

Estos dos atributos de las acciones atómicas nos llevan a dos técnicas adicionales para evitar interferencia: exclusión mutua y sincronización por condición. Consideremos las proofs outlines:

(2.17) *P1* :: ... { *pre(a)* } *a* ....

P2 :: ... S1 { C } S2 ...

Aquí, *a* es una sentencia de asignación en el proceso *P1*, y S1 y S2 son sentencias en el proceso *P2*. Supongamos que *a* interfiere con la aserción crítica C. Una manera de evitar interferencia es usar exclusión mutua para “ocultar” C en *a*. Esto se hace construyendo una única acción atómica:

⟨ S1; S2 ⟩

que ejecuta S1 y S2 atómicamente y hace a C invisible para los otros procesos.

Otra manera de eliminar interferencia en (2.17) es usar sincronización por condición para fortalecer la precondición de *a*. El requerimiento de no interferencia (2.9) será satisfecho si:

* C es falso cuando *a* se ejecuta, y por lo tanto el proceso P2 no podría estar listo para ejecutar S2; o

* la ejecución de *a* hace true a C, es decir, si *post(a)* ⇒ C.

Así, podemos reemplazar *a* en (2.17) por la siguiente acción atómica condicional:

⟨ **await not** C **or** B → *a* ⟩

Aquí, B es un predicado que caracteriza un conjunto de estados tales que ejecutar *a* hará true a C. De la definición de la precondición weakest, B = *wp*(*a*, C) caracteriza el mayor conjunto de tales estados.

Para ilustrar estas técnicas, consideremos el siguiente ejemplo de una sistema bancario simplificado. Supongamos que un banco tiene un conjunto de cuentas, les permite a los clientes transferir dinero de una cuenta a otra, y tiene un auditor para chequear malversaciones. Representamos las cuentas con account[1:n]. Una transacción que transfiere $100 de la cuenta x a la y podría ser implementada por un proceso *Transfer*, donde asumimos que x≠y, que hay fondos suficientes, y que ambas cuentas son válidas. En las aserciones de *Transfer*, X e Y son variables lógicas.

```
var account[1:n] : int
Transfer :: { account[x] = X ∧ account[y] = Y }
            ⟨ account[x] := account[x] - 100
```

```
                account[y] := account[y] + 100 ⟩
              { account[x] = X - 100 ∧ account[y] = Y + 100 }
Auditor :: var total := 0, i :=1, embezzle := false
          { total = account[1] + .... account[i-1] }
          do i ≤ n → { C1: total = account[1] + .... account[i-1]  ∧ i ≤ n }
                       total := total + account[i]; i := i + 1
                       { C2: total = account[1] + .... account[i-1] }
          od
          { total = account[1] + .... account[n] ∧ i = n }
          if total ≠ CASH → embezzle := true fi
```

Sea CASH el monto total en todas las cuentas. El proceso *Auditor* chequea malversaciones iterando a través de las cuentas, sumando los montos en cada una, y luego comparando el total con CASH.

Como está programado en el ejemplo, *Transfer* se ejecuta como una sola acción atómica, de modo que *Auditor* no verá un estado en el cual account[x] fue debitada y luego termine de auditar sin ver el crédito (pendiente) en account[y]. Pero esto no es suficiente para prevenir interferencia. El problema es que, si *Transfer* ejecuta mientras el índice *i* en *Auditor* está entre x e y, el monto viejo en una de las cuentas ya fue sumado a total, y más tarde la nueva cantidad en la otra cuenta va a ser sumada, dejando que el *Auditor* crea incorrectamente que hubo malversación. Más precisamente, las asignaciones en *Transfer* interfieren con las aserciones C1 y C2 de *Auditor*.

Se necesita sincronización adicional para evitar interferencia entre los procesos del ejemplo. Una aproximación es usar exclusión mutua para ocultar las aserciones críticas C1 y C2. Esto se hace poniendo entre corchetes el loop **do** entero en *Auditor* convirtiéndolo en atómico. Desafortunadamente, esto tiene el efecto de hacer que casi todo el *Auditor* ejecute sin interrupción. Una alternativa es usar sincronización por condición para evitar que *Transfer* ejecute si al hacerlo interfiere con C1 y C2. En particular, podemos reemplazar la acción atómica incondicional en *Transfer* por la siguiente acción atómica condicional:

```
⟨ await ( x < i and y < i ) or ( x > i and y > i ) →
    account[x] := account[x] - 100; account[y] := account[y] + 100⟩
```

Esta aproximación tiene el efecto de hacer que *Transfer* se demore solo cuando *Auditor* está en un punto crítico entre las cuentas x e y.

Como ilustra el ejemplo, la exclusión mutua y la sincronización por condición pueden ser usadas *siempre* para evitar interferencia. Más aún, con frecuencia son requeridas dado que las otras técnicas por si mismas son insuficientes. En consecuencia, usaremos mucho la sincronización en combinación con otras técnicas, especialmente invariantes globales. Sin embargo, la sincronización tiene que ser usada con cuidado dado que implica overhead y puede llevar a deadlock si una condición de demora nunca se convierte en true. Afortunadamente, es posible resolver problemas de sincronización en una manera eficiente y sistemática.

VARIABLES AUXILIARES

La lógica de programación PL extendida con la regla de concurrencia (2.11) no es aún una lógica (relativamente) completa. Algunas triplas son válidas pero no se puede probar que lo sean. El problema es que con frecuencia necesitamos hacer aserciones explícitas acerca de los valores de los contadores de programa de los distintos procesos. Sin embargo los program counters son parte del *estado oculto*: valore que no están almacenados en variables de programa. Otros ejemplos de estado oculto son las colas de procesos bloqueados y las colas de mensajes que fueron enviados pero no recibidos.

Para ilustrar el problema y motivar su solución, consideremos el siguiente programa:

(2.18) **co** P1: ⟨ x := x + 1 ⟩ // P2: ⟨ x := x + 1 ⟩ **oc**

Supongamos que x es inicialmente 0. Entonces x será 2 cuando el programa termina. Pero cómo podemos probarlo? En el programa muy similar (2.12), en el cual un proceso incrementaba x en 1 y el otro en 2, pudimos debilitar aserciones para contar con el hecho de que los procesos ejecutaran en cualquier orden. Sin embargo, esa técnica no funciona aquí. Aisladamente, cada una de las siguientes es una tripla válida:

{ x=0 } P1: x := x + 1 { x=1 }
{ x=0 } P2: x := x + 1 { x=1 }

Sin embargo, P1 interfiere con ambas aserciones de la segunda tripla, y P2 interfiere con las de la primera. Para que P2 ejecutara antes que P1, podemos debilitar las aserciones en P1:

{ x=0 $\vee$ x=1 } P1: x := x + 1 { x=1 $\vee$ x=2 }

Pero P1 aún interfiere con las aserciones de P2. Podemos tratar de debilitar las aserciones en P2:

{ x=0 $\vee$ x=1 } P2: x := x + 1 { x=1 $\vee$ x=2 }

Desafortunadamente aún tenemos interferencia, y debilitar más las aserciones no ayudará. Además, las aserciones ya son demasiado débiles para concluir que x=2 en el estado final del programa dado que la conjunción de las postcondiciones es (x=1 $\vee$ x=2).

En (2.12) el hecho de que los procesos incrementaran x en distintos valores proveía suficiente información para distinguir cuál proceso ejecutaba primero. Sin embargo, en (2.18) ambos procesos incrementan x en 1, y entonces el estado x=1 no codifica cuál ejecuta primero. En consecuencia, el orden de ejecución debe ser codificado explícitamente. Esto requiere introducir variables adicionales. Sean t1 y t2 variables agregadas a (2.18) de la siguiente forma:

(2.19) **var** x := 0, t1 := 0, t2 := 0
co P1: ⟨ x := x + 1 ⟩; t1 := 1 // P2: ⟨ x := x + 1 ⟩; t2 := 1 **oc**

El proceso P1 setea t1 en 1 para indicar que incrementó x; P2 usa t2 de manera similar.

En el estado inicial de (2.19), x=t1+t2. El mismo predicado es true en el estado final. Así, si este predicado fuera un invariante global, podríamos concluir que x es 2 en el estado final dado que t1 y t2 son ambos 1 en ese estado. Pero x=t1+t2 no es un invariante global pues no es true justo luego de que cada proceso incrementó x. Sin embargo, podemos ocultar este estado combinando las dos asignaciones en cada proceso en una única acción atómica:

(2.20) **var** x := 0, t1 := 0, t2 := 0 { I: x=t1+t2 }
{ I $\wedge$ t1=0 $\wedge$ t2=0 }
co P1: { I $\wedge$ t1=0 } ⟨ x := x + 1; t1 := 1 ⟩ { I $\wedge$ t1=1 }
// P2: { I $\wedge$ t2=0 } ⟨ x := x + 1; t2 := 1 ⟩ { I $\wedge$ t2=1 }
oc
{ I $\wedge$ t1=1 $\wedge$ t2=1 }

La proof outline para cada proceso es válida aisladamente. Dado que cada aserción es la conjunción del invariante global y un predicado sobre una variable no referenciada por los otros procesos, los procesos están libres de interferencia. Así, (2.20) es una proof outline válida.

El programa (2.20) no es el mismo que (2.18) ya que contiene dos variables extra. Sin embargo, t1 y t2 son *variables auxiliares*. Fueron agregadas al programa sólo para registrar suficiente información de estado para poder construir una prueba. En particular, t1 y t2 cumplen la siguiente restricción:

(2.21) **Restricción de Variable Auxiliar.** Las variables auxiliares aparecen solo en sentencias de asignación x := e donde x es una variable auxiliar.

Una variable auxiliar no puede aparecer en asignaciones a variables de programa o en guardas en sentencias **if**, **do**, y **await**. Por lo tanto no pueden afectar la ejecución del programa al que se agregan. Esto significa que el programa sin variables auxiliares tiene las mismas propiedades de correctitud parcial (y total) que el programa con las variables auxiliares. La siguiente regla establece esto.

Sean P y Q predicados que no referencian variables auxiliares. Sea la sentencia S obtenida a partir de la sentencia S' borrando todas las sentencias que asignan variables auxiliares. Entonces, la siguiente regla de inferencia es sound:

(2.22) **Regla de Variable Auxiliar**. $\dfrac{\{ P \}\ S' \{ Q \}}{\{ P \}\ S \{ Q \}}$

Puede usarse esta regla para probar (2.18).

PROPIEDADES DE SEGURIDAD Y VIDA

Recordemos que una propiedad de un programa es un atributo que es verdadero en cada posible historia de ese programa. Toda propiedad interesante puede ser formulada en términos de dos clases de propiedades: seguridad y vida. Una propiedad de seguridad asegura que nada malo ocurre durante la ejecución; una propiedad de vida afirma que algo bueno eventualmente ocurre. En los programas secuenciales, la propiedad de seguridad clave es que el estado final es correcto, y la clave de la propiedad de vida es terminación. Estas propiedades son igualmente importantes para programas concurrentes. Además, hay otras propiedades interesantes de seguridad y vida que se aplican a los programas concurrentes.

Dos propiedades de seguridad importantes en programas concurrentes son exclusión mutua y ausencia de deadlock. Para exclusión mutua, lo malo es tener más de un proceso ejecutando secciones críticas de sentencias al mismo tiempo. Para ausencia de deadlock, lo malo es tener algunos procesos esperando condiciones que no ocurrirán nunca.

Ejemplos de propiedades de vida de programas concurrentes son que una pedido de servicio eventualmente será atendido, que un mensaje eventualmente alcanzará su destino, y que un proceso eventualmente entrará a su sección crítica. Las propiedades de vida están afectadas por las políticas de scheduling, las cuales determinan cuales acciones atómicas elegibles son las próximas en ejecutarse.

Prueba de propiedades de seguridad

Todas las acciones que toma un programa deben estar basadas en su estado. Aún las acciones realizadas en respuesta a leer una entrada están basadas en el estado del programa dado que los valores de entrada disponibles par un programa pueden pensarse como parte del estado. Así, si un programa falla al satisfacer esta propiedad de seguridad, debe haber alguna secuencia de estados (historia) que fallan al satisfacer esta propiedad.

Para las propiedades de seguridad que nos interesan (por ejemplo, corrección parcial, exclusión mutua, y ausencia de deadlock) debe haber algún estado de programa individual que falla al satisfacer esta propiedad. Por ejemplo, si la propiedad de exclusión mutua falla, debe haber algún estado en el cual dos (o más) procesos están simultáneamente en sus secciones críticas.

Para las propiedades de seguridad que pueden ser especificadas por la ausencia de un mal estado de programa, hay un método simple para probar que un programa satisface la propiedad. Sea BAD un predicado que caracteriza un estado de programa malo. Entonces un programa satisface la propiedad de seguridad asociada si BAD no es true en ningún estado del programa. Dado el programa S, para mostrar que BAD no es true en ningún estado se requiere mostrar que no es true en el estado inicial, en el segundo estado, y así siguiendo, donde el estado cambia como resultado de ejecutar acciones atómicas elegibles. Las aserciones críticas en una prueba { P } S { Q } caracterizan los estados inicial, intermedio y final. Esto provee la base para el siguiente método de prueba de que un programa satisface una propiedad de seguridad.

(2.23) **Prueba de una Propiedad de Seguridad**. Sea BAD un predicado que caracteriza un estado de programa malo. Asumimos que { P } S { Q } es una prueba en PL y que la precondición P caracteriza el estado inicial del programa. Entonces S satisface la propiedad de seguridad especificada por $\neg$BAD si, para toda aserción crítica C en la prueba, C $\Rightarrow \neg$BAD.

P debe caracterizar el estado inicial del programa para descartar la prueba trivial en la cual toda aserción crítica es la constante *false*.

Recordemos que un invariante global *I* es un predicado que es true en cualquier estado visible de un programa concurrente. Esto sugiere un método alternativo para probar una propiedad de seguridad.

(2.24) **Prueba de una Propiedad de Seguridad usando un Invariante**. Sea BAD un predicado que caracteriza un estado de programa malo. Asumimos que { P } S { Q } es una prueba en PL, que P caracteriza el estado inicial del programa, y que *I* es un invariante global en la prueba. Entonces S satisface la propiedad de seguridad especificada por $\neg$BAD si *I* $\Rightarrow \neg$BAD.

Muchas propiedades de seguridad interesantes pueden ser formuladas en términos de predicados que caracterizan estados en que los procesos no deberían estar simultáneamente. Por ejemplo, en exclusión mutua, las precondiciones de las secciones críticas de los dos procesos no deberían ser simultáneamente verdaderas. Si el predicado P caracteriza algún estado de un proceso y el predicado Q caracteriza algún estado de un segundo proceso, entonces P $\wedge$ Q será true si ambos procesos están en sus respectivos estados. Además, si (P $\wedge$ Q) = false, entonces los procesos no pueden estar simultáneamente en los dos estados ya que ningún estado satisface false.

Supongamos que una propiedad de seguridad puede ser caracterizada por BAD = (P$\wedge$Q) y que P y Q no son simultáneamente true en un programa dado; es decir, (P$\wedge$Q)=false. Entonces el programa satisface la propiedad de seguridad. Esto es porque $\neg$BAD = $\neg$(P$\wedge$Q)=true, y cualquier aserción crítica en una prueba implicará esto trivialmente. Esto lleva al siguiente método muy útil para probar una propiedad de seguridad.

(2.25) **Exclusión de Configuraciones**. Dada una prueba de un programa, un proceso no puede están en un estado que satisface P mientras otro proceso está en un estado que satisface Q si (P$\wedge$Q)=false.

Como ejemplo del uso de (2.25), consideremos la proof outline del programa que copia el arreglo. La sentencia **await** en cada proceso puede causar demora. El proceso podría quedar en deadlock si ambos estuvieran demorados y ninguno pudiera proceder. El proceso *Productor* es demorado si está en su sentencia **await** y la condición de demora es falsa; en ese estado, el siguiente predicado sería true:

$$PC \wedge p < n \wedge p \neq c$$

De manera similar, el proceso *Consumidor* es demorado si está en su sentencia **await** y la condición de demora es falsa; ese estado satisface:

$$IC \wedge c < n \wedge p \leq c$$

Dado que la conjunción de estos dos predicados es falsa, los procesos no pueden estar simultáneamente en estos estados; por lo tanto no puede ocurrir deadlock.

Políticas de Scheduling y Fairness

La mayoría de las propiedades de vida dependen de *fairness*, la cual trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los otros procesos. Recordemos que una acción atómica en un proceso es elegible si es la próxima acción atómica en el procesos que será ejecutado. Cuando hay varios procesos, hay varias acciones atómicas elegibles. Una *política de scheduling* determina cuál será la próxima en ejecutarse.

Dado que las acciones atómicas pueden ser ejecutadas en paralelo solo si no interfieren, la ejecución paralela puede ser modelizada por ejecución serial, interleaved. Por lo tanto para definir los atributos formales de las políticas de scheduling, enfatizamos en esta sección el scheduling en un solo procesador.

Una política de scheduling de bajo nivel, tal como la política de alocación de procesador en un sistema operativo, concierne a la performance y utilización del hardware. Esto es importante, pero igualmente importante son los atributos globales de las políticas de scheduling y sus efecto sobre la terminación y otras propiedades de vida de los programas concurrentes. Consideremos el siguiente programa con dos procesos, *Loop* y *Stop*:

var continue := true
Loop :: **do** continue $\rightarrow$ **skip od**
Stop :: continue := false

Supongamos una política de scheduling que asigna un procesador a un proceso hasta que el proceso termina o se demora. Si hay un solo procesador, el programa anterior no terminará si *Loop* se ejecuta primero. Sin embargo, el programa terminará si eventualmente *Stop* tiene chance de ejecutar.

(2.26) **Fairness Incondicional**. Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada.

Para el programa anterior, round-robin sería una política incondicionalmente fair en un único procesador, y la ejecución paralela sería incondicionalmente fair en un multiprocesador.

Cuando un programa contiene acciones atómicas condicionales, necesitamos hacer suposiciones más fuertes para garantizar que los procesos progresarán. Esto es porque una acción atómica condicional, aún si es elegible, es demorada hasta que la guarda es true.

(2.27) **Fairness Débil**. Una política de scheduling es débilmente fair si es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada si su guarda se convierte en true y de allí en adelante permanece true.

En síntesis, si $\langle$ **await** B $\rightarrow$ S $\rangle$ es elegible y B se vuelve true y permanece true, entonces la acción atómica eventualmente se ejecuta. Round-robin y timeslicing son políticas débilmente fair si todo proceso tiene chance de ejecutar. Esto es porque cualquier proceso demorado eventualmente verá que su condición de demora es true.

Sin embargo, esto no es suficiente para asegurar que cualquier sentencia **await** elegible eventualmente se ejecuta. Esto es porque la guarda podría cambiar el valor (de false a true y nuevamente a false) mientras un proceso está demorado. En este caso, necesitamos una política de scheduling más fuerte.

(2.28) **Fairness Fuerte**. Una política de scheduling es fuertemente fair si es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada si su guarda es true con infinita frecuencia.

Una guarda es true con infinita frecuencia si es true un número infinito de veces en cada historia de ejecución de un programa (non-terminating). Para ser fuertemente fair, una política no puede considerar seleccionar solo una acción cuando la guarda es false; debe seleccionar alguna vez la acción cuando la guarda es true.

Para ver la diferencia entre las políticas débiles y fuertes, consideremos el siguiente programa:

(2.29) **var** continue := true, try := false
Loop :: **do** continue → try := true; try := false **od**
Stop :: ⟨ **await** try → continue := false ⟩

Con una política fuertemente fair, este programa eventualmente terminará ya que try es true con infinita frecuencia. Sin embargo, con una política débilmente fair podría no terminar ya que try es también false con infinita frecuencia.

Desafortunadamente, es imposible tener una política de scheduling de procesador general que sea práctica y fuertemente fair. Consideremos el programa (2.29) nuevamente. En un solo procesador, un scheduler que alterna las acciones de los dos procesos sería fuertemente fair ya que *Stop* vería un estado en el cual try es true; pero, tal scheduler es impráctico de implementar. Por otro lado, round-robin y timeslicing son prácticos pero no fuertemente fair en general. Un scheduler multiprocesador que ejecuta los procesos en (2.29) en paralelo es también práctico, pero no es fuertemente fair. En los últimos casos, *Stop* podría siempre examinar try cuando es falso. Por supuesto esto es difícil, pero teóricamente posible.

Sin embargo, hay instancias especiales de políticas fuertemente fair y prácticas. Por ejemplo, supongamos dos procesos que ejecutan repetidamente

⟨ **await** s > 0 → s := s - 1 ⟩

y que otros procesos repetidamente incrementan s. También supongamos que los dos procesos son scheduled en orden FCFS (es decir, cada vez que ambos están tratando de ejecutar **await**, el que estuvo esperando más tiempo es scheduled primero). Entonces cada proceso progresará continuamente. FCFS es una instancia especial de round-robin.

Para clarificar las distintas clases de políticas de scheduling, consideremos nuevamente el programa que copia un arreglo. Como vimos, ese programa está libre de deadlock. Así, el programa terminará si cada proceso tiene chance de progresar. Cada uno lo hará ya que la política es débilmente fair. Esto es porque, cuando un proceso hace true la condición de demora del otro, esa condición se mantiene true hasta que otro proceso continue y cambie las variables compartidas.

Ambas sentencias **await** en ese programa tienen la forma ⟨ **await** B ⟩, y B se refiere a solo una variable alterada por el otro proceso. En consecuencia, ambas sentencias **await** pueden ser implementadas por loops busy waiting. Por ejemplo, ⟨ **await** p=c ⟩ en el *Productor* puede ser implementada por:

do p ≠ c → **skip od**

En este caso, el programa terminará si la política de scheduling es incondicionalmente fair dado que no hay acciones atómicas condicionales y los procesos se alternan el acceso al buffer compartido. Sin embargo, este no es en general el caso en que una política incondicionalmente fair asegurará terminación de un loop busy waiting; por ejemplo, ver (2.29). Esto es porque una política incondicionalmente fair podría siempre schedule la acción atómica que examina la guarda del loop cuando la guarda es true. Cuando un loop busy waiting nunca termina, un programa se dice que sufre *livelock*, la analogía de deadlock. La ausencia de livelock es una propiedad de vida (la cosa buena sería la eventual terminación del loop) dado que con busy waiting un proceso siempre tiene alguna acción que puede ejecutar.

Variables Compartidas

Los loops son las sentencias más complejas en los programas secuenciales. Como ya describimos, la clave para desarrollar y entender un loop es encontrar un invariante. Esto involucra enfocar qué es lo que no cambia cada vez que el loop se ejecuta. El mismo punto de vista puede usarse para resolver problemas de sincronización, que son los problemas claves en los programas concurrentes.

El rol de la sincronización es evitar interferencia, o haciendo conjuntos de sentencias atómicas y ocultando estados intermedios (exclusión mutua) o demorando un proceso hasta que se de una condición (sincronización por condición). Siempre podemos evitar interferencia usando sentencias **await**, pero estas no siempre pueden ser implementadas eficientemente. En esta parte mostraremos cómo usar primitivas de sincronización de más bajo nivel que pueden ser implementadas eficientemente en hardware o software. Usaremos las técnicas ya introducidas (variables disjuntas, aserciones debilitadas, invariantes globales).

Un propósito es ilustrar un método para resolver problemas de sincronización de manera sistemática. La base del método es ver a los procesos como *mantenedores de invariantes*. En particular, para cada problema de sincronización definimos un invariante global que caracteriza relaciones claves entre variables locales y globales. Con respecto a sincronización, el rol de cada proceso es asegurar que se mantiene el invariante. También particionaremos variables para resolver distintos problemas en conjuntos disjuntos, cada uno con su propio invariante. El método de solución consiste de 4 pasos:

1. **Definir el problema en forma precisa**. Identificar los procesos y especificar el problema de sincronización. Introducir las variables necesarias y escribir un predicado que especifica la propiedad invariante que debe mantenerse.

2. **"Outline" de la solución**. Escribir en los procesos las asignaciones a variables. Inicializar las variables para que el invariante sea inicialmente verdadero. Encerrar entre corchetes las secuencias de asignaciones cuando deben ser ejecutadas atómicamente.

3. **Asegurar el invariante**. Cuando sea necesario, poner guardas a las acciones atómicas incondicionales para asegurar que cada acción mantiene el invariante global. Dada la acción S y el invariante *I*, esto involucra computar *wp(S,I)* para determinar el conjunto de estados a partir de los cuales la ejecución de S garantiza que termina con *I* verdadero. Dado que S es una secuencia de sentencias de asignación, terminará una vez que empezó.

4. **Implementar las acciones atómicas**. Transformar las acciones atómicas resultantes de los pasos 2 y 3 en código que emplea sólo sentencias secuenciales y primitivas de sincronización disponibles.

Los primeros tres pasos son esencialmente independientes de las primitivas de sincronización disponibles. El punto de partida es dar con un invariante apropiado. Dado que todas las propiedades de seguridad pueden ser expresadas como un predicado sobre los estados de programa, esto sugiere dos maneras de encontrar un invariante: especificar un predicado BAD que caracterice un estado malo y luego usar ¬BAD como invariante global; o directamente especificar un predicado GOOD que caracteriza estados buenos y usarlo como invariante.

Dado un invariante, el segundo y tercer paso son mecánicos. El paso creativo es el primero, que involucra usar algún mecanismo de sincronización específico para implementar las acciones atómicas.

Sincronización Fine-Grained

Recordemos que *busy waiting* es una forma de sincronización en la cual un proceso chequea repetidamente una condición hasta que es verdadera. La ventaja es que podemos implementarlo usando solo las instrucciones de máquina disponibles en cualquier procesador. Aunque es ineficiente cuando los procesos se ejecutan por multiprogramación, puede ser aceptable y eficiente cuando cada proceso se ejecuta en su propio procesador. Esto es posible en multiprocesadores. El hardware en sí mismo emplea sincronización por busy waiting; por ejemplo, se usa para sincronizar transferencias de datos sobre buses de memoria y redes locales.

Este capitulo examina problemas prácticos y muestra cómo resolverlos usando busy waiting. El primer problema es el de la sección crítica. Se desarrollan 4 soluciones con distintas técnicas y distintas propiedades. Las soluciones a este problema son importantes pues pueden usarse para implementar sentencias **await** y por lo tanto acciones atómicas arbitrarias.

También se examina una clase de programas paralelos y un mecanismo de sincronización asociado. Muchos problemas pueden resolverse con algoritmos iterativos paralelos en los cuales varios procesos manipulan repetidamente un arreglo compartido. Esta clase de algoritmos se llama *algoritmo paralelo de datos* ya que los datos compartidos se manipulan en paralelo. En tal algoritmo, cada iteración depende típicamente de los resultados de la iteración previa. Luego, al final de cada iteración, cada proceso necesita esperar a los otros antes de comenzar la próxima iteración. Esta clase de punto de sincronización se llama *barrier*. También se describen multiprocesadores sincrónicos (SIMD), especiales para implementar algoritmos paralelos de datos. Esto es porque los SIMD ejecutan instrucciones en lock step en cada procesador, entonces proveen sincronización por barrier automático.

EL PROBLEMA DE LA SECCION CRITICA

El problema de la *sección crítica* es uno de los problemas clásicos de la programación concurrente. Fue el primer problema estudiado y mantiene su interés dado que las soluciones pueden usarse para implementar sentencias **await**.

En este problema, *n* procesos repetidamente ejecutan una sección crítica de código, luego una sección no crítica. La sección crítica está precedida por un protocolo de entrada y seguido de un protocolo de salida.

(3.1) P[i:1..n] :: **do** true →
entry protocol
critical section
exit protocol
non-critical section
od

Cada sección crítica es una secuencia de sentencias que acceden algún objeto compartido. Cada sección no crítica es otra secuencia de sentencias. Asumimos que un proceso que entra en su sección crítica eventualmente sale; así, un proceso solo puede finalizar fuera de su sección crítica. Nuestra tarea es diseñar protocolos de entrada y salida que satisfagan las siguientes propiedades:

(3.2) **Exclusión mutua**. A lo sumo un proceso a la vez está ejecutando su sección crítica

(3.3) **Ausencia de Deadlock**. Si dos o más procesos tratan de entrar a sus secciones críticas, al menos uno tendrá éxito.

(3.4) **Ausencia de Demora Innecesaria**. Si un proceso está tratando de entrar a su SC y los otros están ejecutando sus SNC o terminaron, el primero no está impedido de entrar a su SC.

(3.5) **Eventual Entrada**. Un proceso que está intentando entrar a su SC eventualmente lo hará.

La primera propiedad es de seguridad, siendo el estado malo uno en el cual dos procesos están en su SC. En una solución busy-waiting, (3.3) es una propiedad de vida llamada *ausencia de livelock*. Esto es porque los procesos nunca se bloquean (están vivos) pero pueden estar siempre en un loop tratando de progresar. (Si los procesos se bloquean esperando para entrar a su SC, (3.3) es una propiedad de seguridad). La tercera propiedad es de seguridad, siendo el estado malo uno en el cual el proceso uno no puede continuar. La última propiedad es de vida ya que depende de la política de scheduling.

En esta sección se desarrolla una solución que satisface las primeras tres propiedades. Esto es suficiente para la mayoría de las aplicaciones ya que es poco probable que un proceso no pueda eventualmente entrar a su SC.

Una manera trivial de resolver el problema es encerrar cada SC en corchetes angulares, es decir, usar **await** incondicionales. La exclusión mutua se desprende automáticamente de la semántica de los corchetes angulares. Las otras tres propiedades deberían satisfacerse si la política de scheduling es incondicionalmente fair ya que ésta asegura que un proceso que intenta ejecutar la acción atómica correspondiente a su SC eventualmente lo hará, sin importar qué hicieron los otros procesos. Sin embargo, esta “solución” trae el tema de cómo implementar los corchetes angulares.

Una solución Coarse-Grained

Para el problema de la SC, las 4 propiedades son importantes, pero la más crítica es la de la exclusión mutua. Para especificar la propiedad de exclusión mutua, necesitamos tener alguna manera de indicar que un proceso está dentro de su SC. Así, necesitamos introducir variables adicionales. Para simplificar la notación, desarrollamos una solución para dos procesos; rápidamente se generaliza a *n* procesos.

Sean in1 e in2 variables booleanas. Cuando el proceso P1 está en su SC, in1 es true; en otro caso, in1 es falsa. El proceso P2 e in2 están relacionadas de la misma manera. Podemos especificar la propiedad de exclusión mutua por:

MUTEX: $\neg(in1 \wedge in2)$

Este predicado tiene la forma de una propiedad de seguridad, donde lo malo es que in1 e in2 sean true a la vez. Agregando estas nuevas variables a (3.1) tenemos el siguiente outline de solución:

```
var in1 := false, in2 := false
{ MUTEX: ¬(in1 ∧ in2) }
P1 :: do true → in1 := true          # entry protocol
                critical section
                in1 := false         # exit protocol
                non-critical section
   od
P2 :: do true → in2 := true          # entry protocol
                critical section
                in2 := false         # exit protocol
                non-critical section
   od
```

Este programa no resuelve el problema. Para asegurar que MUTEX es invariante, necesitamos que sea true antes y después de cada asignación a in1 o in2. A partir de los valores iniciales asignados a las variables, MUTEX es inicialmente true. Consideremos ahora el entry protocol en P1. Si MUTEX es true después de esta asignación, antes de la asignación el estado debe satisfacer:

$wp(in1 := true, MUTEX) = \neg(true \wedge in2) = \neg in2$

Luego, necesitamos fortalecer el entry protocol en P1 reemplazando la asignación incondicional por la acción atómica condicional:

⟨ **await not** in2 → in1 := true ⟩

Los procesos son simétricos, entonces usamos la misma clase de acción atómica condicional para el entry protocol de P2.

Qué sucede con los protocolos de salida? En general, nunca es necesario demorar cuando se deja la SC. Más formalmente, si MUTEX debe ser true luego de que el protocolo de salida de P1, la precondición debe cumplir:

wp(in1 := false, MUTEX) = ¬(false ∧ in2) = true

Esto, por supuesto, es satisfecho por cualquier estado. La misma situación existe para el protocolo de P2. Entonces, no necesitamos guardar los protocolos de salida.

Reemplazando los protocolos de entrada por acciones atómicas condicionales tenemos la solución coarse-grained y proof outline siguiente:

```
var in1 := false, in2 := false
{ MUTEX: ¬(in1 ∧ in2) }
P1 :: do true → { MUTEX ∧ ¬in1 }
                ⟨ await not in2 → in1 := true ⟩   # entry protocol
                { MUTEX ∧ in1 }
                critical section
                in1 := false                      # exit protocol
                { MUTEX ∧ ¬in1 }
                non-critical section
      od
P2 :: do true → { MUTEX ∧ ¬in2 }
                ⟨ await not in1 → in2 := true ⟩   # entry protocol
                { MUTEX ∧ in2 }
                critical section
                in2 := false                      # exit protocol
                { MUTEX ∧ ¬in2 }
                non-critical section
      od
```

Por construcción, la solución satisface la propiedad de exclusión mutua. La ausencia de deadlock y demora innecesaria se deduce del método de Exclusión de Configuraciones (2.25). Si los procesos están en deadlock, cada uno está tratando de entrar a su SC pero no lo puede hacer. Esto significa que las precondiciones de los protocolos de entrada son ambas true, pero ninguna guarda es true. Así, el siguiente predicado caracteriza un estado de deadlock:

¬in1 ∧ in2 ∧ ¬in2 ∧ in1

Como este predicado es false, no puede ocurrir deadlock en el programa anterior.

Consideremos la ausencia de demora innecesaria. Si el proceso P1 está fuera de su SC o terminó, entonces in1 es false. Si P2 está tratando de entrar a su SC pero no puede hacerlo, entonces la guarda en su entry protocol debe ser falsa. Dado que

¬in1 ∧ in1 = false

P2 no puede ser demorado innecesariamente. Algo análogo sucede con P1.

Finalmente, consideremos la propiedad de vida de que un proceso que intenta entrar a su SC eventualmente es capaz de hacerlo. Si P1 está tratando de entrar pero no puede, P2 está en su SC e in2 es true. Como suponemos que un proceso que está en su SC eventualmente sale, in2 se volverá falsa y la guarda de entrada de P1 será verdadera. Si P1 aún no puede entrar, es porque el scheduler es unfair o porque P2 volvió a ganar la entrada. En la última situación, la historia se repite, e in2 en algún momento se vuelve false. Así, in2 es true con infinita frecuencia (o P2 se detiene, en cuyo caso in2 se convierte y permanece true. Una política de scheduling strongly fair es suficiente para asegurar que P1 va a entrar. La argumentación para P2 es simétrica.

Spin Locks: Una solución Fine-Grained

La solución coarse-grained vista emplea dos variables. Para generalizar la solución a *n* procesos, deberíamos usar *n* variables. Pero, hay solo dos estados que nos interesan: algún proceso está en su SC o ningún proceso lo está. Una variable es suficiente para distinguir entre estos dos estados, independiente del número de procesos.

Para este problema, sea *lock* una variable booleana que indica cuando un proceso está en su SC. O sea que *lock* es true cuando in1 o in2 lo son:

lock = (in1 $\vee$ in2)

Usando *lock* en lugar de in1 e in2, tenemos:

```
var lock := false
P1 :: do true → ⟨ await not lock → lock := true ⟩   # entry protocol
                critical section
                lock := false                        # exit protocol
                non-critical section
   od
P2 :: do true → ⟨ await not lock → lock := true ⟩   # entry protocol
                critical section
                lock := false                        # exit protocol
                non-critical section
   od
```

El significado de este cambio de variables es que casi todas las máquinas (especialmente multiprocesadores) tienen alguna instrucción especial que puede usarse para implementar las acciones atómicas condicionales de este programa (test-and-set, fetch-and-add, compare-and-swap). Por ahora definimos y usamos Test-and-Set (TS):

La instrucción TS toma dos argumentos booleanos: un *lock* compartido y un código de condición local *cc*. Como una acción atómica, TS setea *cc* al valor de *lock*, luego setea *lock* a true:

(3.6) TS(lock,cc): ⟨ cc := lock; lock := true ⟩

Usando TS podemos implementar la solución coarse-grained anterior:

```
var lock := false
P1 :: var cc : bool
   do true → TS(lock,cc)                             # entry protocol
             do cc → TS(lock,cc) od
             critical section
             lock := false                             # exit protocol
             non-critical section
   od
P2 :: var cc : bool
   do true → TS(lock,cc)                             # entry protocol
             do cc → TS(lock,cc) od
             critical section
             lock := false                             # exit protocol
```

```
                    non-critical section
            od
```

Reemplazamos las acciones atómicas condicionales en la solución coarse-grained por loops que no terminan hasta que lock es false, y por lo tanto TS setea cc a falso. Si ambos procesos están tratando de entrar a su SC, solo uno puede tener éxito en ser el primero en setear lock en true; por lo tanto, solo uno terminará su entry protocol. Cuando se usa una variable de lockeo de este modo, se lo llama *spin lock* pues los procesos “dan vueltas” (spin) mientras esperan que se libere el lock.

Los dos programas anteriores resuelven correctamente el problema de la SC. Se asegura la exclusión mutua pues solo uno de los procesos puede ver que lock es false. La ausencia de deadlock resulta del hecho de que, si ambos procesos están en sus entry protocols, *lock* es false, y entonces uno de los procesos tendrá éxito para entrar a su SC. Se evita la demora innecesaria porque, si ambos procesos están fuera de su SC, *lock* es false, y por lo tanto uno puede entrar si el otro está ejecutando su SNC o terminó. Además, un proceso que trata de entrar a su SC eventualmente tendrá éxito si el scheduling es fuertemente fair pues *lock* será true con infinita frecuencia.

La última solución tiene un atributo adicional que no tiene la primera solución planteada (con in1 e in2): resuelve el problema para cualquier número de procesos, no solo para dos. Esto es porque hay solo dos estados de interés, independiente del número de procesos.

Una solución al problema de la SC similar a la última puede emplearse en cualquier máquina que tenga alguna instrucción que testea y altera una variable compartida como una única acción atómica. Por ejemplo, algunas máquinas tienen una instrucción de incremento que incrementa un valor entero y también toma un código de condición indicando si el resultado es positivo o negativo. Usando esta instrucción, el entry protocol puede basarse en la transición de cero a uno. Algo a tener en cuenta al construir una solución busy waiting al problema de la SC es que en la mayoría de los casos el protocolo de salida debería retornar las variables compartidas a su estado inicial.

Aunque la última solución es correcta, se demostró que en multiprocesadores puede llevar a baja performance si varios procesos están compitiendo por el acceso a una SC. Esto es porque *lock* es una variable compartida y todo proceso demorado continuamente la referencia. Esto causa “memory contention”, lo que degrada la performance de las unidades de memoria y las redes de interconexión procesador-memoria.

Además, la instrucción TS escribe en *lock* cada vez que es ejecutada, aún cuando el valor de *lock* no cambie. Dado que la mayoría de los multiprocesadores emplean caches para reducir el tráfico hacia la memoria primaria, esto hace a TS mucho más cara que una instrucción que solo lee una variable compartida (al escribir un valor en un procesador, deben invalidarse o modificarse los caches de los otros procesadores). El overhead por invalidación de cache puede reducirse modificando el entry protocol para usar un protocolo test-and-test-and-set como sigue:

```
(3.7)   do lock → skip od              # spin mientras se setea lock
        TS(lock,cc)
        do cc → do lock → skip od      # nuevamente spin
                TS(lock,cc)
        od
```

Aquí, un proceso solamente examina *lock* hasta que hay posibilidad de que TS pueda tener éxito, Como *lock* solo es examinada en los dos loops adicionales, su valor puede ser leido desde un cache local sin afectar a los otros procesadores. Sin embargo, memory contention sigue siendo un problema. Cuando *lock* es limpiada, al menos uno y posiblemente todos los procesos demorados ejecutarán TS, aunque solo uno puede proseguir. La próxima sección presenta una manera de atacar este problema.

Implementación de Sentencias Await

Cualquier solución al problema de la SC puede usarse para implementar una acción atómica incondicional ⟨ S ⟩ ocultando puntos de control internos a los otros procesos. Sea CSenter un entry protocol a una SC, y CSexit el correspondiente exit protocol. Entonces ⟨ S ⟩ puede ser implementada por:

```
CSenter
S
CSexit
```

Esto asume que las SC en todos los procesos que examinan variables alteradas en S también están protegidas por entry y exit protocols similares. En esencia, ⟨ es reemplazada por CSenter, y ⟩ por CSexit.

El esqueleto de código anterior puede usarse como building block para implementar cualquier acción atómica condicional ⟨ **await** B → S ⟩. Recordemos que una acción atómica condicional demora al proceso hasta que B es true, luego ejecuta S. También, B debe ser true cuando comienza la ejecución de S. Para asegurar que la acción entera es atómica, podemos usar un protocolo de SC para ocultar los estados intermedios en S. Luego podemos usar un loop para testear B repetidamente hasta que sea true. Luego, el esqueleto de código para implementar ⟨ **await** B → S ⟩ es:

```
CSenter
do not B → ? od
S
CSexit
```

Asumimos que las SC en todos los procesos que alteran variables referenciadas en B o S o que referencian variables alteradas en S están protegidas por protocolos similares.

Lo que resta es ver cómo implementar el cuerpo del loop. Si el cuerpo se ejecuta, B era falsa. Luego, la única manera en que B se volverá true es si algún otro proceso altera una variable referenciada en B. Como asumimos que cualquier sentencia en otro proceso que altera una variable referenciada en B debe están en una SC, tenemos que salir de la SC mientras esperamos que B se vuelva true. Pero para asegurar la atomicidad de la evaluación de B y la ejecución de S, debemos reentrar a la SC antes de reevaluar B. Un refinamiento del anterior código es:

(3.8)
```
CSenter
do not B → CSexit; CSenter od
S
CSexit
```

La implementación preserva la semántica de las acciones atómicas condicionales, asumiendo que los protocolos de SC garantizan exclusión mutua. Si el scheduling es débilmente fair, el proceso que ejecuta (3.8) eventualmente terminará el loop, asumiendo que B se convierte en true y permanece true. Este tipo de scheduling también es suficiente para asegurar entrada eventual a una SC. Si el scheduling es fuertemente fair, el loop se terminará si B se convierte en true con infinita frecuencia.

Aunque (3.8) es correcta, es ineficiente. Esto es porque un proceso está “spinning” en un “hard loop” (continuamente saliendo y entrando a la SC) aunque posiblemente no podrá proceder hasta que al menos algún otro proceso altere una variable referenciada en B. Esto lleva a memory contention ya que cada proceso demorado accede continuamente las variables usadas en los protocolos de SC y las variables de B.

Para reducir el problema, es preferible para un proceso demorarse algún período antes de reentrar a la SC. Sea Delay algún código que “enlentece” a un proceso. Podemos reemplazar (3.8) por el siguiente protocolo para implementar una acción atómica condicional:

(3.9)
```
CSenter
```

do not B → CSexit; Delay; CSenter **od**
S
CSexit

El código de Delay podría, por ejemplo, ser un loop vacío que itera un número aleatorio de veces. Este clase de protocolo back-off también es útil dentro de los protocolos CSenter en si mismos; por ej, puede ser agregado al loop de demora en el entry protocolo test-and-set.

Si S es la sentencia **skip**, el protocolo (3.9) puede simplificarse omitiendo S. Si además B satisface los requerimientos de la propiedad de A Lo Sumo Una Vez (2.5), la sentencia ⟨ **await** B ⟩ puede implementarse como:

do not B → **skip od**

Esta implementación también es suficiente si B permanece true una vez que se convirtió en true.

Como mencionamos al comienzo, la sincronización busy waiting con frecuencia es usada dentro del hardware. De hecho, un protocolo similar a (3.9) se usa en los controladores Ethernet para sincronizar el acceso a una LAN. En particular, para transmitir un mensaje, un controlador Ethernet primero lo envía, luego escucha para ver si colisionó con otro mensaje de otro controlador. Si no hay colisión, se asume que la transmisión fue exitosa. Si se detecta una colisión, el controlador se demora, y luego intenta reenviar el mensaje. Para evitar una race condition en la cual dos controladores colisionan repetidamente, la demora es elegida aleatoriamente en un intervalo que es doblado cada vez que ocurre una colisión. Por lo tanto, esto es llamado protocolo “binary exponential back-off”. Este tipo de protocolos es útil en (3.9) y en entry protocols de SC.

SECCIONES CRITICAS: ALGORITMO TIE-BREAKER

Cuando una solución al problema de la SC emplea una instrucción como Test-and-Set, el scheduling debe ser fuertemente fair para asegurar la eventual entrada. Este es un requerimiento fuerte pues las políticas de scheduling prácticas son solo débilmente fair. Aunque es improbable que un proceso que trata de entrar a su SC nunca tenga éxito, podría suceder si dos o más procesos están siempre compitiendo por la entrada. Esto es porque la solución spin lock no controla el orden en el cual lo procesos demorados entran a sus SC si dos o más están tratando de hacerlo.

El *algoritmo tie-breaker* (o algoritmo de Peterson) es un protocolo de SC que requiere solo scheduling incondicionalmente fair para satisfacer la propiedad de eventual entrada. Además no requiere instrucciones especiales del tipo Test-and-Set. Sin embargo, el algoritmo es mucho más complejo que la solución spin lock.

Solución Coarse-Grained

Consideremos nuevamente el programa coarse-grained usando in1 e in2. Ahora el objetivo es implementar las acciones atómicas condicionales usando solo variables simples y sentencias secuenciales. Por ejemplo, queremos implementar el entry protocol en el proceso P1,

⟨ **await not** in2 → in1 := true ⟩

en términos de acciones atómicas fine-grained.

Como punto de partida, consideremos implementar cada sentencia **await** primero quedándonos en un loop hasta que la guarda sea true, y luego ejecutando la asignación. El entry protocol para P1 sería:

do **in2 → skip od** # entry protocol para P1

```
in1 := true
```

Análogamente, el entry protocol para P2 sería:

```
do in1 → skip od          # entry protocol para P1
in2 := true
```

El exit protocol para P1 setearía in1 en false, y el de P2, in2 en false.

El problema con esta "solución" es que las dos acciones en los entry protocols no se ejecutan atómicamente, por lo que no podemos asegurar exclusión mutua. Por ejemplo, la postcondición deseada para el loop de demora en P1 es que in2 es falso. Desafortunadamente, esto es interferido por la asignación in2:=true. Operacionalmente es posible para ambos procesos evaluar sus condiciones de demora casi al mismo tiempo y encontrar que son true.

Dado que cada proceso quiere estar seguro que el otro no está en su SC cuando el **do** termina, consideremos cambiar el orden de las sentencias en los entry protocols:

```
in1 := true               # entry protocol para P1
do in2 → skip od

in2 := true               # entry protocol para P2
do in1 → skip od
```

Esto ayuda pero aún no resuelve el problema. Se asegura exclusión mutua pero puede haber deadlock: Si in1 e in2 son ambas true, ningún loop va a terminar. Sin embargo, hay una manera simple de evitar deadlock: Usar una variable adicional para romper el empate si ambos procesos están demorados.

Sea *last* una variable entera que indica cuál de P1 y P2 fue el último en comenzar a ejecutar su entry protocol. Luego, si P1 y P2 están tratando de entrar a sus SC (in1 e in2 son true) el último proceso en comenzar su entry protocol es demorado. Esto lleva a la siguiente solución coarse-grained:

```
var in1 := false; in2 := false; last := 1
P1:: do true → in1 := true; last := 1          # entry protocol
                ⟨ await not in2 or last = 2 ⟩
                critical section
                in1 := false                   # exit protocol
                non-critical section
     od
P2:: do true → in2 := true; last := 2          # entry protocol
                ⟨ await not in1 or last = 1 ⟩
                critical section
                in2 := false                   # exit protocol
                non-critical section
     od
```

Solución Fine-Grained

El algoritmo anterior está muy cercano a una solución fine-grained que no requiere sentencias **await**. En particular, si cada **await** satisficiera los requerimientos de la propiedad de a-lo-sumo-una-vez (2.4), podría ser implementada por loops busy-waiting. Desafortunadamente, cada **await** referencia dos variables alteradas por otro proceso. Sin embargo, en este caso no es necesario que las condiciones de demora sean evaluadas atómicamente. Informalmente esto es verdad por las siguientes razones.

Consideremos las sentencias **await** en P1. Si la condición de demora es true, o in2 es false o last es 2. La única manera que P2 podría hacer false la condición de demora es si ejecuta la primera sentencia de su entry protocol, la cual setea in2 a true. Pero entonces la próxima acción de P2 es hacer true nuevamente la condición seteando last a 2. Así, cuando P1 está en su SC, puede asegurarse que P2 no estará en su SC. El argumento para P2 es simétrico.

Dado que las condiciones de demora no necesitan ser evaluadas atómicamente, cada **await** puede ser reemplazada por un loop **do** que itera mientras la negación de la condición de demora es false. Esto lleva al siguiente algoritmo tie-breaker fine-grained:

```
var in1 := false; in2 := false; last := 1
P1:: do true → in1 := true; last := 1          # entry protocol
                do in2 and last = 1 → skip od
                critical section
                in1 := false                   # exit protocol
                non-critical section
     od
P2:: do true → in2 := true; last := 2          # entry protocol
                do in1 and last = 2 → skip od
                ⟨ await not in1 od last = 1 ⟩
                critical section
                in2 := false                   # exit protocol
                non-critical section
     od
```

Para probar formalmente que el algoritmo es correcto, podemos introducir dos variables auxiliares para registrar cuando cada proceso está entre las dos primeras asignaciones en su entry protocol. Por ejemplo, podemos reemplazar las dos primeras asignaciones en P1 por:

```
⟨ in1 := true; mid1 := true ⟩
⟨ last := 1; mid1 := false ⟩
```

Haciendo un cambio similar en P2, la siguiente aserción es verdadera cuando P1 está en su SC:

$$\{ in1 \wedge \neg mid1 \wedge (\neg in2 \vee last=2 \vee mid2) \}$$

Con esta y la aserción correspondiente antes de la SC de P2, podemos usar Exclusión de Configuraciones (2.25) para concluir que P1 y P2 no pueden ejecutar sus SC simultáneamente.

Solución N-Proceso

El anterior algoritmo tie-breaker resuelve el problema de la SC para dos procesos. Podemos usar la idea básica para resolver el problema para cualquier número de procesos. En particular, si hay *n* procesos, el entry protocol en cada proceso consiste de un loop que itera a través de n-1 etapas. En cada etapa, usamos instancias del algoritmo tie-breaker para dos procesos para determinar cuales procesos avanzan a la siguiente etapa. Si aseguramos que a lo sumo a un proceso a la vez se le permite ir a través de las n-1 etapas, entonces a lo sumo uno a la vez puede estar en su SC.

Sean in[1:n] y last[1:n] arreglos enteros, donde n>1. El valor de in[i] indica cuál etapa está ejecutando p[i]; el valor de last[j] indica cuál proceso fue el último en comenzar la etapa j. Estas variables son usadas de la siguiente manera:

```
var in[1:n] := ( [n] 0 ); last[1:n] := ( [n] 0 )
P[i: 1..n] :: do true →
                  fa j := 1 to n-1 →       # entry protocol
                     # registra que el proceso i está en la etapa j y es el último
                     in[i] := j; last[j] := i
                     fa k := 1 to n st i ≠ k →
```

```
            # espera si el proceso k está en una etapa mayor y
            # el proceso i fue el último en entrar a esta etapa
            do in[k] ≥ in[i] and last[j] = i → skip od
        af
    af
    critical section
    in[i] := 0              # exit protocol
    non-critical section
od
```

El for-all externo se ejecuta n-1 vez. El for-all interno en el proceso P[i] chequea los otros procesos. En particular, P[i] espera si hay algún otro proceso en una etapa numerada igual o mayor y P[i] fue el último proceso en entrar a la etapa j. Una vez que otro proceso entra a la etapa j o todos los procesos “adelante” de P[i] dejaron su SC, P[i] puede pasar a la siguiente etapa. Así, a lo sumo n-1 procesos pueden haber pasado la primera etapa, n-2 la segunda, etc. Esto asegura que a lo sumo un proceso a la vez puede completar las n-1 etapas y por lo tanto estar ejecutando su SC.

La solución n-proceso está libre de livelock, evita demora innecesaria, y asegura eventual entrada. Estas propiedades se desprenden del hecho de que un proceso se demora solo si algún otro proceso está adelante de él en el entry protocol, y de la suposición de que todo proceso sale de su SC.

En esta solución, el entry protocol ejecuta $O(n^2)$ instancias del algoritmo tie-breaker para dos procesos. Esto es porque el for-all interno es ejecutado n-1 veces en cada una de las n-1 iteraciones del loop externo. Esto sucede aún si solo un proceso está tratando de entrar a su SC. Hay una variación del algoritmo que requiere ejecutar solo O(n*m) instancias del algoritmo tie-breaker si solo m procesos están compitiendo para entrar a la SC. Sin embargo, el otro algoritmo tiene una varianza mucho mayor en tiempo de demora potencial cuando hay contención.

SECCIONES CRITICAS: ALGORITMO TICKET

El algoritmo tie-breaker n-proceso es bastante complejo y difícil de entender. Esto es en parte porque no es obvio cómo generalizar el algoritmo de 2 procesos a n. Desarrollaremos una solución al problema de la SC para n-procesos que es mucho más fácil de entender. La solución también ilustra cómo pueden usarse contadores enteros para ordenar procesos. El algoritmo se llama *ticket algorithm* pues se basa en repartir tickets (números) y luego esperar turno.

Solución Coarse-Grained

Algunos negocios emplean el siguiente método para asegurar que los clientes son servidos en orden de llegada. Luego de entrar al negocio, un cliente toma un número que es mayor que el tomado por cualquier otro. El cliente luego espera hasta que todos los clientes con un número más chico sean atendidos. Este algoritmo es implementado por un repartidor de números y por un display que indica qué cliente está siendo servido. Si el negocio tiene un empleado, los clientes son servidos uno a la vez en orden de llegada. Podemos usar esta idea para implementar un protocolo de SC fair.

Sean *number* y *next* enteros inicialmente 1, y sea *turn*[1:n] un arreglo de enteros, cada uno de los cuales es inicialmente 0. Para entrar a su SC, el proceso P[i] primero setea *turn*[i] al valor corriente de *number* y luego incrementa *number*. Son acciones atómicas simples para asegurar que los clientes obtienen números únicos. El proceso P[i] luego espera hasta que el valor de *next* es igual a su número. En particular, queremos que el siguiente predicado sea invariante:

$$\{ \text{TICKET: } (\text{P[i] está en su SC}) \Rightarrow (\text{turn[i]} = \text{next}) \wedge (\forall i,j : 1 \le i, j \le n, i \ne j: \text{turn[i]} = 0 \vee \text{turn[i]} \ne \text{turn[j]}) \}$$

El segundo conjuntor dice que los valores de *turn* que no son cero son únicos; entonces a lo sumo un *turn*[i] es igual a *next*. Luego de completar su SC, P[i] incrementa *next*, nuevamente como una acción atómica. El protocolo resulta en el siguiente algoritmo:

```
var number := 1, next := 1, turn[1:n] : int := ( [n] 0 )
{ TICKET: ( P[i] está en su SC) ⇒ ( turn[i] = next) ∧
          ( ∀i,j : 1 ≤ i, j ≤ n, i ≠ j: turn[i] = 0 ∨ turn[i] ≠ turn[j] ) }
P[i: 1..n] :: do true →
                 ⟨ turn[i] := number; number := number + 1 ⟩
                 ⟨ await turn[i] := next ⟩
                 critical section
                 ⟨ next := next + 1 ⟩
                 non-critical section
              od
```

El predicado TICKET es un invariante global ya que *number* es leído e incrementado como una acción atómica y *next* es incrementada como una acción atómica. Por lo tanto a lo sumo un proceso puede estar en su SC. La ausencia de deadlock y demora innecesaria se desprenden del hecho de que los valores distintos de cero en *turn* son únicos. Finalmente, si el scheduling es débilmente fair, el algoritmo asegura entrada eventual ya que una vez que una condición de demora se vuelve verdadera, permanece verdadera.

A diferencia del algoritmo tie-breaker, el algoritmo ticket tiene un problema potencial que es común en algoritmos que emplean incrementos en contadores: los valores de *number* y *next* son ilimitados. Si el algoritmo corre un tiempo largo, se puede alcanzar un overflow. Para este algoritmo podemos resolver el problema reseteando los contadores a un valor chico (digamos 1) cada vez que sean demasiado grandes. Si el valor más grande es al menos tan grande como n, entonces los valores de turn[i] se garantiza que son únicos.

Solución Fine-Grained

El algoritmo anterior emplea tres acciones atómicas coarse-grained. Es fácil implementar la sentencia **await** usando un loop busy-waiting ya que la expresión booleana referencia solo una variable compartida. Aunque la última acción atómica (que incrementa *next*) referencia *next* dos veces, también puede ser implementada usando instrucciones load y store regulares. Esto es porque a lo sumo un proceso a la vez puede ejecutar el protocolo de salida. Desafortunadamente, es difícil en general implementar la primera acción atómica, la cual lee *number* y luego la incrementa.

Algunas máquinas tienen instrucciones que retornan el viejo valor de una variable y la incrementan o decrementan como una operación indivisible simple. Esta clase de instrucción hace exactamente lo que requiere el algoritmo ticket. Como ejemplo específico, Fetch-and-Add es una instrucción con el siguiente efecto:

```
FA(var,incr): ⟨ temp := var; var := var + incr; return(temp) ⟩
```

El siguiente es el algoritmo ticket implementado usando FA. (Como en la solución coarse-grained, podemos evitar overflow reseteando los contadores cuando alcanzan un límite mayor que n).

```
var number := 1, next := 1, turn[1:n] : int := ( [n] 0 )
{ TICKET: ( P[i] está en su SC) ⇒ ( turn[i] = next) ∧
          ( ∀i,j : 1 ≤ i, j ≤ n, i ≠ j: turn[i] = 0 ∨ turn[i] ≠ turn[j] ) }
P[i: 1..n] :: do true →
                 turn[i] := FA(number,1)
                 do turn[i] ≠ next → skip od
                 critical section
                 next := next + 1
                 non-critical section
              od
```

En máquinas que no tienen una instrucción FA o similar, tenemos que usar otra aproximación. El requerimiento clave en el algoritmo ticket es que cada proceso obtenga un número único. Si una máquina tiene una instrucción de incremento atómica, podríamos considerar implementar el primer paso en el entry protocol por:

turn[i] := number; ⟨ number := number + 1 ⟩

Esto asegura que *number* es incrementada correctamente, pero no asegura que los procesos obtengan números únicos. En particular, cada proceso podría ejecutar la primera asignación casi al mismo tiempo y obtener el mismo número. Así, es esencial que ambas asignaciones sean ejecutadas como una acción atómica simple.

Ya vimos otras dos maneras de resolver el problema de la SC: spin locks y el algoritmo tie-breaker. Cualquiera de estas podría usarse dentro del algoritmo ticket para hacer atómica la obtención de número. En particular, sea CSenter un entry protocol de SC, y CSexit el correspondiente exit protocol. Entonces podríamos reemplazar la sentencia FA por:

(3.10) CSenter; turn[i] := number; number := number + 1; CSexit

Aunque esta podría parecer una aproximación curiosa, en la práctica funcionaría bastante bien, especialmente si se dispone de una instrucción como Test-and-Set para implementar CSenter y CSexit. Con Test-and-Set, los procesos podrían obtener los números no en exactamente el orden que intentan (y teóricamente un proceso podría quedarse dando vueltas para siempre) pero con probabilidad muy alta cada proceso obtendría un número, y la mayoría en orden. Esto es porque la SC dentro de (3.10) es muy corta, y por lo tanto un proceso no se demoraría en CSenter. La mayor fuente de demora en el algoritmo ticket es esperar a que turn[i] sea igual a *next*.

SECCIONES CRITICAS: ALGORITMO BAKERY

El algoritmo ticket puede ser implementado directamente en máquinas que tienen una instrucción como Fetch-and-Add. Si solo tenemos disponibles instrucciones menos poderosas, podemos simular la parte de obtención del número del algoritmo ticket usando (3.10). Pero eso requiere usar otro protocolo de SC, y la solución podría no ser fair. Presentaremos un algoritmo del tipo de ticket (llamado *bakery algorithm*) que es fair y no requiere instrucciones de máquina especiales. El algoritmo es más complejo que el ticket, pero ilustra una manera de romper empates cuando dos procesos obtienen el mismo número.

Solución Coarse-Grained

En el algoritmo ticket, cada cliente obtiene un número único y luego espera a que su número sea igual a *next*. El algoritmo bakery no requiere un “despachante” de números atómico y no usa un contador *next*. En particular, cuando un cliente entra al negocio, primero mira alrededor a todos los otros clientes y setea su número a uno mayor a cualquiera que ve. Luego espera a que su número sea menor que el de cualquier otro cliente. Como en el algoritmo ticket, el cliente con el número más chico es el próximo en ser servido. La diferencia es que los clientes se chequean uno con otro en lugar de con un contador central *next* para decidir el orden de servicio.

Como en el algoritmo ticket, sea *turn*[1:n] un arreglo de enteros, cada uno de los cuales es inicialmente 0. Para entrar a su SC, el proceso P[i] primero setea *turn*[i] a uno más que el máximo de los otros valores de *turn*. Luego P[i] espera hasta que *turn*[i] sea el más chico de los valores no nulos de *turn*. Así, el algoritmo bakery mantiene invariante el siguiente predicado:

BAKERY: (P[i] está ejecutando su SC) $\Rightarrow$ (turn[i] $\neq$ 0 $\wedge$
($\forall$,j : 1 $\leq$ j $\leq$ n, j $\neq$ i: turn[j] = 0 $\vee$ turn[i] < turn[j]))

Luego de completar su SC, P[i] resetea *turn*[i] a 0.

La siguiente es una solución coarse-grained del algoritmo bakery:

var turn[1:n] : **int** := ([n] 0)
{ BAKERY: (P[i] está ejecutando su SC) $\Rightarrow$ (turn[i] $\neq$ 0 $\wedge$
($\forall$,j : 1 $\leq$ j $\leq$ n, j $\neq$ i: turn[j] = 0 $\vee$ turn[i] < turn[j])) }
P[i: 1..n] :: **do** true $\rightarrow$

```
        ⟨ turn[i] := max(turn[1:n]) + 1 ⟩
        fa j := 1 to n st j ≠ i →
                ⟨ await turn[j] = 0 or turn[i] < turn[j] ⟩
        af
        critical section
        ⟨ turn[i] := 0 ⟩
        non-critical section
    od
```

La primera acción atómica garantiza que los valores no nulos de *turn* son únicos. La sentencia for-all asegura que el consecuente en el predicado BAKERY es true cuando P[i] está ejecutando su SC. El algoritmos satisface la propiedad de exclusión mutua pues turn[i] ≠ 0, turn[j] ≠ 0, y BAKERY no pueden ser todos verdaderos a la vez. No puede haber deadlock pues los valores no nulos de *turn* son únicos, y como es usual asumimos que cada proceso eventualmente sale de su SC. Los procesos no son demorados innecesariamente pues turn[i] es 0 cuando P[i] está fuera de su SC. Finalmente, el algoritmo asegura entrada eventual si el scheduling es débilmente fair pues una vez que una condición de demora se convierte en true, permanece true.

Los valores de *turn* en el algoritmo bakery pueden volverse arbitrariamente grandes. A diferencia del algoritmo ticket, este problema no puede ser resuelto "ciclando" sobre un conjunto finito de enteros. Sin embargo, turn[i] sigue agrandándose solo si *siempre* hay al menos un proceso tratando de entrar a su SC. Este no es un problema práctico pues significa que los procesos están gastando demasiado tiempo para entrar a sus SC. En este caso, es inapropiado usar busy waiting.

Solución Fine-Grained

El algoritmo bakery coarse-grained no puede ser implementado directamente en máquinas contemporáneas. La asignación a turn[i] requiere computar el máximo de n valores, y la sentencia **await** referencia una variable compartida dos veces. Estas acciones podrían ser implementadas atómicamente usando otro protocolo de SC tal como el algoritmo tie-breaker, pero sería bastante ineficiente. Afortunadamente, hay una aproximación más simple.

Cuando *n* procesos necesitan sincronizar, con frecuencia es útil desarrollar una solución para n = 2 y luego generalizar esa solución. Este fue el caso para el algoritmo tie-breaker y nuevamente es útil aquí ya que ayuda a ilustrar los problemas que hay que resolver. Consideremos la siguiente versión de dos procesos del algoritmo bakery coarse-grained:

(3.11)
```
var turn1 := 0, turn2 := 0
P1 :: do true → turn1 := turn2 + 1
                do turn2 ≠ 0 and turn1 > turn2 → skip od
                critical section
                turn1 := 0
                non-critical section
      od
P2 :: do true → turn2 := turn1 + 1
                do turn1 ≠ 0 and turn2 > turn1 → skip od
                critical section
                turn2 := 0
                non-critical section
      od
```

Aquí, cada proceso setea su valor de *turn* con una versión optimizada de (3.10), y las sentencias **await** son implementadas tentativamente por un loop busy-waiting.

El problema con esta "solución" es que ni las sentencias de asignación en los entry protocols ni las guardas del **do** loop satisfacen la propiedad de A-lo-sumo-una-vez (2.4), por lo que no serán evaluadas atómicamente. En consecuencia, los procesos podrían comenzar sus entry protocols casi al mismo tiempo, y ambos podrían setear turn1 y turn2 a 1. Si ocurre esto, ambos procesos podrían estar en su SC al mismo tiempo.

El algoritmo tie-breaker para dos procesos sugiere una solución parcial al problema de (3.11): si turn1 y turn2 son ambos 1, se deja a uno de los procesos seguir y se demora al otro. Por ejemplo, dejemos seguir al proceso de menor número fortaleciendo el segundo conjuntor en el loop de demora en P2 a turn2 $\geq$ turn1.

Desafortunadamente, aún es posible para ambos procesos entrar a su SC. Por ejemplo, supongamos que P1 lee turn2 y obtiene 0. Luego supongamos que P2 comienza su entry protocol, ve que turn1 aún es 0, setea turn2 a 1, y luego entra a su SC. En este punto, P1 puede continuar su entry protocol, setea turn1 a 1, y luego entra a su SC pues turn1 y turn2 son 1 y P1 tiene precedencia en este caso. Esta clase de situación es llamada *race condition* pues P2 "raced by" P1 y por lo tanto P1 se perdió ver que P2 estaba cambiando turn2.

Para evitar esta race condition, podemos hacer que cada proceso setee su valor de turn a 1 (o cualquier otro valor no nulo) al comienzo de su entry protocol. Luego examina los otros valores de turn y resetea el suyo. La solución es la siguiente:

```
var turn1 := 0, turn2 := 0
P1 :: do true → turn1 := 1; turn1 := turn2 + 1
                do turn2 ≠ 0 and turn1 > turn2 → skip od
                critical section
                turn1 := 0
                non-critical section
      od
P2 :: do true → turn2 := 1; turn2 := turn1 + 1
                do turn1 ≠ 0 and turn2 ≥ turn1 → skip od
                critical section
                turn2 := 0
                non-critical section
      od
```

Un proceso no puede salir de su loop **do** hasta que el otro terminó de setear su valor de turn si está en el medio de hacer esto. La solución da a P1 precedencia sobre P2 en caso de que ambos tengan el mismo valor (no nulo) para turn. Cuando P1 está en su SC, el siguiente predicado es true:

$$turn1 > 0 \wedge (turn2 = 0 \vee turn1 \leq turn2)$$

Similarmente, cuando P2 está en su SC,

$$turn2 > 0 \wedge (turn1 = 0 \vee turn2 < turn1)$$

La exclusión mutua de las SC se desprende del método de Exclusión de Configuraciones (2.25) ya que la conjunción de las precondiciones de las SC es false. El algoritmo bakery para dos procesos también satisface las otras propiedades de la SC.

Los procesos en la solución anterior no son simétricos ya que las condiciones de demora en el segundo loop son apenas diferentes. Sin embargo, podemos reescribirlas en una forma simétrica como sigue. Sean (a,b) y (c,d) pares de enteros, y definamos la relación *mayor que* entre tales pares como sigue:

```
(a,b) > (c,d) = true    si a > c o si a = c y b > d
                false   en otro caso
```

Luego podemos reescribir turn1 > turn2 en P1 como (turn1,1) > (turn2,2) y podemos reescribir turn2 $\geq$ turn1 en P2 como (turn2,2) > (turn1,1).

La virtud de una especificación simétrica es que ahora es fácil generalizar el algoritmo bakery de dos procesos para n procesos:

```
var turn[1:n] : int := ( [n] 0 )
{ BAKERY: ( P[i] está ejecutando su SC) ⇒ ( turn[i] ≠ 0  ∧
```

```
                ( ∀,j : 1 ≤ j ≤ n, j ≠ i: turn[j] = 0 ∨ turn[i]  < turn[j]
                        ∨ (turn[i] = turn[j]  ∧  i < j) ) ) }
P[i: 1..n] :: var j : int
            do true →
               turn[i] := 1; turn[i] := max(turn[1:n]) + 1
               fa j := 1 to n st j ≠ i →
                        do turn[j] ≠ 0 and (turn[i],i) > (turn[j],j) → skip od
               af
               critical section
               turn[i] := 0
               non-critical section
            od
```

La solución emplea un loop for-all como en la solución coarse-grained de modo que un proceso se demora hasta que tiene precedencia sobre todos los otros. El predicado BAKERY es un invariante global que es casi idéntico al invariante global de la solución coarse-grained. La diferencia está en la tercera línea, la cual refleja el hecho de que, en la solución fine-grained, dos procesos podrían tener el mismo valor para su elemento de *turn*. Aquí se da precedencia al proceso de índice menor.

SINCRONIZACION BARRIER

Muchos problemas pueden ser resueltos usando algoritmos iterativos que sucesivamente computan mejores aproximaciones a una respuesta, terminando o cuando la respuesta final fue computada o (en el caso de muchos algoritmos numéricos) cuando la respuesta final ha convergido. Típicamente tales algoritmos manipulan un arreglo de valores, y cada iteración realiza la misma computación sobre todos los elementos del arreglo. Por lo tanto, podemos usar múltiples procesos para computar partes disjuntas de la solución en paralelo.

Un atributo clave de la mayoría de los algoritmos iterativos paralelos es que cada iteración típicamente depende de los resultados de la iteración previa. Una manera de estructurar tal algoritmo es implementar el cuerpo de cada iteración usando una o mas sentencias **co**. Ignorando terminación, y asumiendo que hay *n* tareas paralelas en cada iteración, esta aproximación tiene la forma general:

```
do true →
    co i := 1 to n → código para implementar tarea i oc
od
```

Desafortunadamente, esta aproximación es bastante ineficiente dado que **co** produce *n* procesos en cada iteración. Es mucho más costoso crear y destruir procesos que implementar sincronización entre procesos. Así, una estructura alternativa resultará en un algoritmo más eficiente. En particular, crear los procesos una vez al comienzo de la computación, y luego sincronizarlos al final de cada iteración:

```
Worker[i: 1..n] :: do true →
                       código para implementar la tarea i
                       esperar a que se completen las n tareas
                   od
```

Este tipo de sincronización es llamada *barrier synchronization* dado que el punto de demora al final de cada iteración representa una barrera a la que todos los procesos deben arribar antes de que se les permita pasar.

Desarrollaremos varias implementaciones busy-waiting de sincronización barrier. Cada una emplea una técnica distinta de interacción entre procesos. También se describe cuándo es apropiado usar cada clase de barrera.

Contador Compartido

La manera más simple de especificar los requerimientos para una barrera es emplear un entero compartido, *count*, el cual es inicialmente 0. Asumimos que hay *n* procesos worker que necesitan encontrarse en una barrera. Cuando un proceso llega a la barrera, incrementa *count*. Por lo tanto, cuando *count* es *n*, todos los procesos pueden continuar. Para especificar esto precisamente, sea *passed*[i] una variable booleana que inicialmente es false; Worker[i] setea *passed*[i] a true cuando ha pasado la barrera. Entonces la propiedad de que ningún worker pase la barrera hasta que todos hayan arribado es asegurada si el siguiente predicado es un invariante global:

COUNT :: ($\forall$ i : 1 $\leq$ i $\leq$ n : passed[i] $\Rightarrow$ count = n)

Si los procesos usan *count* y *passed* como definimos, y guardando las asignaciones a *passed* para asegurar que COUNT es invariante, tenemos la siguiente solución parcial:

(3.12)
```
var count := 0, passed[1:n] : bool := ( [n] false )
Worker[i: 1..n] :: do true →
                       código para implementar la tarea i
                       ⟨ count := count + 1 ⟩
                       ⟨ await count = n → passed[i] := true ⟩
                   od
```

Aquí, *passed* es una variable auxiliar que es usada solo para especificar la propiedad de barrera. Después de borrarla del programa, podemos implementar la sentencia **await** por un loop busy-waiting. También, muchas máquinas tienen una instrucción de incremento indivisible. Por ejemplo, usando la instrucción Fetch-and-Add podemos implementar la barrera anterior por:

```
FA(count,1)
do count ≠ n → skip od
```

Pero el programa anterior no resuelve totalmente el problema. La dificultad es que *count* debe ser 0 al comienzo de cada iteración. Por lo tanto, *count* necesita ser reseteada a 0 cada vez que todos los procesos han pasado la barrera. Más aún, tiene que ser reseteada antes de que cualquier proceso trate nuevamente de incrementar *count*.

Es posible resolver este problema de “reset” empleando dos contadores, uno que cuenta hasta *n* y otro que cuenta hacia abajo hasta 0, con los roles de los contadores switcheados después de cada etapa. Sin embargo, hay problemas adicionales, pragmáticos, con el uso de contadores compartidos. Primero, tienen que ser incrementados y/o decrementados como acciones atómicas. Segundo, cuando un proceso es demorado en (3.12), está examinando continuamente *count*. En el peor caso, *n* - 1 procesos podrían estar demorados esperando que el n-ésimo proceso llegue a la barrera. Esto podría llevar a memory contention, excepto en multiprocesadores con caches coherentes. Pero aún así, el valor de *count* está cambiando continuamente, por lo que cada cache necesita ser actualizado. Así, es apropiado implementar una barrera usando contadores solo si la máquina destino tiene instrucciones de incremento atómicas, caches coherentes, y actualización de cache eficiente. Más aún, *n* debería ser relativamente chico (a lo sumo 30).

Flags y Coordinadores

Una manera de evitar el problema de contención de memoria es distribuir la implementación de *count* usando *n* variables que sumen el mismo valor. En particular, sea *arrive*[1:n] un arreglo de enteros inicializado en 0. Reemplacemos el incremento de *count* en (3.12) por *arrive*[i] := 1. Con este cambio, el siguiente predicado es un invariante global:

(3.13) count = arrive[1] + ... + arrive[n]

La contención de memoria se evita si los elementos de *arrive* son almacenados en distintos bancos de memoria.

Con el cambio anterior, los problemas que restan son implementar la sentencia **await** en (3.12) y resetear los elementos de *arrive* al final de cada iteración. Usando la relación (3.13) e ignorando la variable auxiliar *passed*, la sentencia **await** puede ser implementada como:

⟨ **await** (arrive[1] + ... + arrive[n]) = n ⟩

Sin embargo, esto reintroduce memory contention. Además es ineficiente ya que la suma de los arrive[i] está siendo computada continuamente por cada Worker que está esperando.

Podemos resolver los problemas de contención y reset usando un conjunto adicional de valores compartidos y empleando un proceso adicional, *Coordinator*. En lugar de que cada *Worker* tenga que sumar y testear los valores de *arrive*, hacemos que cada Worker espere que un único valor se convierta en true. En particular, sea *continue*[1:n] otro arreglo de enteros, inicializado en 0. Después de setear *arrive*[i] en 1, Worker[i] se demora esperando que *continue*[i] sea seteada en 1:

(3.14) *arrive*[i] := 1
⟨ **await** continue[i] = 1 ⟩

El proceso *Coordinator* espera a que todos los elementos de *arrive* se vuelvan 1, luego setea todos los elementos de *continue* en 1:

(3.14) **fa** i := 1 **to** n → ⟨ **await** *arrive*[i] = 1 ⟩ **af**
fa i := 1 **to** n → *continue*[i] := 1 **af**

Luego, *arrive* y *continue* tienen la siguiente interpretación:

(∀ i : 1 ≤ i ≤ n : (arrive[i] = 1) ⇒ Worker[i] alcanzó la barrera) ∧
(∀ i : 1 ≤ i ≤ n : (continue[i] = 1) ⇒ Worker[i] puede pasar la barrera)

Las sentencias **await** en (3.14) y (3.15) pueden ser implementadas por **do** loops dado que cada una referencia una única variable compartida. También, el *Coordinator* puede usar una sentencia for-all para esperar que cada elemento de *arrive* sea seteado; dado que todos deben ser seteados antes de que a cualquier *Worker* se le permita continuar, el *Coordinator* puede testear el arreglo *arrive* en cualquier orden. Finalmente, la contención de memoria no es un problema pues los procesos esperan que distintas variables sean seteadas y estas variables podrían estar almacenadas en distintas unidades de memoria.

Las variables *arrive* y *continue* son llamadas *flag variables*. Esto es porque cada variable es alcanzada por un proceso para señalar que una condición de sincronización es true. El problema remanente es aumentar (3.14) y (3.15) con código para limpiar los flags reseteándolos a 0 en preparación para la próxima iteración. Aquí se aplican dos principios generales.

(3.16) **Flag Synchronization Principles**. El proceso que espera a que un flag de sincronización sea seteado es el que debería limpiar el flag. Un flag no debería ser seteado hasta que se sabe que está limpio.

La primera parte del principio asegura que un flag no es limpiado antes de que se vio que fue seteado. Así, en (3.14) Worker[i] debería limpiar *continue*[i], y en (3.15) Coordinator debería limpiar todos los elementos de *arrive*. La segunda parte del principio asegura que otro proceso no puede setear nuevamente el mismo flag antes de que el flag sea limpiado, lo cual podría llevar a deadlock si el primer proceso espera a que el flag sea seteado nuevamente. En (3.15) esto significa que Coordinator debería limpiar *arrive*[i] antes de setear *continue*[i]. El Coordinator puede hacer esto ejecutando otra sentencia for-all después de la primera en (3.15). Alternativamente, Coordinator puede limpiar *arrive*[i] inmediatamente después de que esperó a que sea seteada. Agregando el código para limpiar los flags, tenemos la siguiente solución:

var arrive[1:n] : **int** := ([n] 0), continue[1:n] : **int** := ([n] 0)
Worker[i: 1..n] :: **do** true →
código para implementar la tarea *i*
arrive[i] := 1
⟨ **await** continue[i] = 1 ⟩

```
                continue[i] := 0
            od
Coordinator :: var i : int
        do true →
            fa i := 1 to n → ⟨ await arrive[i] = 1 ⟩; arrive[i] := 0 af
            fa i := 1 to n → continue[i] := 1 af
        od
```

Aunque esto implementa barrier synchronization en una forma que evita la contención de memoria, la solución tiene dos atributos indeseables. Primero, requiere un proceso extra. Dado que la sincronización busy-waiting es ineficiente a menos que cada proceso ejecute en su propio procesador, el Coordinator debería ejecutar en su propio procesador, el cual no está disponible para la ejecución de otro proceso que podría estar haciendo trabajo "útil".

El segundo problema es que el tiempo de ejecución de cada iteración de Coordinator (y por lo tanto cada instancia de barrier synchronization) es proporcional al número de procesos Worker. En algoritmos iterativos, el código ejecutado por cada Worker es típicamente idéntico, y por lo tanto cada uno va a arribar casi al mismo tiempo a la barrera si cada Worker es ejecutado en su propio procesador. Así, todos los flags *arrive* serían seteados casi al mismo tiempo. Sin embargo, Coordinator cicla a través de los flags, esperando que cada uno sea seteado en turno.

Podemos solucionar estos problemas combinando las acciones del coordinador y los workers de modo que cada worker sea también un coordinador. En particular, podemos organizar los workers en un árbol. Entonces podemos hacer que los workers envíen señales de arribo hacia arriba en el árbol y señales de continue hacia abajo. En particular, un nodo worker primero espera a que sus hijos arriben, luego le dice a su nodo padre que él también arribó. Cuando el nodo raíz sabe que sus hijos han arribado, sabe que todos los otros workers también lo han hecho. Por lo tanto la raíz puede decirle a sus hijos que continúen; estos a su vez pueden decirle a sus hijos que continúen, etc. Las acciones específicas de cada clase de proceso worker se ven en el siguiente programa. (Las sentencias **await** pueden ser implementadas por spin loops):

```
leaf node l :  arrive[l] := 1
               ⟨ await continue[l] = 1 ⟩; continue[l] := 0

interior node i :  ⟨ await arrive[left] = 1 ⟩; arrive[left] := 0
                   ⟨ await arrive[right] = 1 ⟩; arrive[right] := 0
                   arrive[i] := 1
                   ⟨ await continue[i] = 1 ⟩; continue[i] := 0
                   continue[left] := 1; continue[right] := 1

root node r : ⟨ await arrive[left] = 1 ⟩; arrive[left] := 0
              ⟨ await arrive[right] = 1 ⟩; arrive[right] := 0
              continue[left] := 1; continue[right] := 1
```

Esta implementación es llamada *combining tree barrier*. Esto es porque cada proceso combina los resultados de sus hijos, y luego se los pasa a su padre. Dado que la altura del árbol es ($\log_2 n$), la aproximación es buena. En particular, usa el mismo número de variables que el coordinador centralizado, pero es mucho más eficiente para un *n* grande.

En multiprocesadores que usan broadcast para actualizar caches, podemos hacer más eficiente esta solución si el nodo raíz hace broadcast de un único mensaje que le dice a todos los otros nodos que continúen. En particular, la raíz setea un flag *continue*, y todos los otros nodos esperan que sea seteado. Este flag *continue* puede luego ser limpiado de dos maneras: Una es usar doble buffering (usar dos flags continue y alternar entre ellos); la otra es alternar el sentido del flag continue (en rondas impares esperar a que sea seteada en 1, y en rondas pares esperar a que sea 0).

Barreras Simétricas

En combining tree barrier, los procesos juegan distintos roles. En particular, los nodos en el interior ejecutan más acciones que las hojas o la raíz. Más aún, el nodo raíz necesita esperar el arribo de señales de arribo para propagarlas hacia arriba en el árbol. Si cada proceso está ejecutando en un procesador diferente y está ejecutando el mismo algoritmo (que es el caso de los algoritmos iterativos paralelos) entonces todos los procesos deberían arribar a la barrera casi al mismo tiempo. Así, si cada proceso toma la misma secuencia de acciones cuando alcanza una barrera, entonces todos podrían ser capaces de seguir en paralelo. Esta sección presenta dos barreras simétricas. Son especialmente adecuadas para multiprocesadores de memoria compartida con tiempo de acceso a memoria no uniforme.

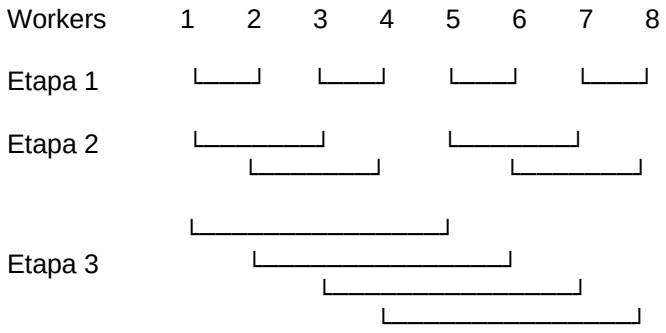
Una barrera simétrica para n procesos se construye a partir de pares de barreras simples para dos procesos. Para construir una barrera de dos procesos, podríamos usar la técnica coordinador/worker. Sin embargo, las acciones de los dos procesos serían diferentes. En lugar de esto, podemos construir una barrera completamente simétrica como sigue. Sea que cada proceso tiene un flag que setea cuando arriba a la barrera. Luego espera a que el otro proceso setee su flag y finalmente limpia la bandera del otro. Si P[i] es un proceso y P[j] es el otro, la barrera simétrica de dos procesos es implementada como sigue:

(3.17)

```
P[i] :: ⟨ await arrive[i] = 0 ⟩          P[i] :: ⟨ await arrive[j] = 0 ⟩
        arrive[i] := 1                           arrive[j] := 1
        ⟨ await arrive[j] = 1 ⟩                  ⟨ await arrive[i] = 1 ⟩
        arrive[j] := 0                           arrive[i] := 0
```

La segunda, tercera y cuarta líneas en cada proceso sigue los principios de flags de sincronización (3.16). La primera línea se necesita para guardar contra un proceso racing back la barrera y seteando su flag antes de que otro proceso limpió su flag.

La pregunta ahora es cómo combinar las barreras para dos procesos para construir una barrera n-proceso. Sean Worker[1:n] los n procesos. Si n es potencia de 2, podríamos combinarlos como muestra la siguiente figura:



Esta clase de barrera se llama *butterfly barrier* debido a la forma del patrón de interconexión, el cual es similar al patrón de interconexión para la transformada de Fourier.

Una butterfly barrier tiene $\log_2 n$ etapas. Cada Worker sincroniza con un Worker distinto en cada etapa. En particular, en la etapa *s* un Worker sincroniza con un Worker a distancia 2^{s-1}. Cada una de las barreras de dos procesos es implementada como muestra la Fig. 3.17 (ver texto), y se usan distintas variables flag para cada barrera de dos procesos. Cuando cada Worker pasó a través de $\log_2 n$ etapas, todos los Workers deben haber arribado a la barrera y por lo tanto todos pueden seguir. Esto es porque cada Worker ha sincronizado directa o indirectamente con cada uno de los otros.

Cuando n no es potencia de 2, puede construirse una butterfly barrier usando la siguiente potencia de 2 mayor que n y teniendo procesos Worker que sustituyan los perdidos en cada etapa. Esto no es muy eficiente. En general, es mejor usar lo que se llama *dissemination barrier*. Nuevamente hay etapas, y en la etapa *s* un Worker sincroniza con uno a distancia 2^{s-1}. Sin embargo, cada barrera de dos procesos es implementada ligeramente diferente a la butterfly barrier. En particular, cada Worker setea su flag de arribo para un Worker a su derecha (módulo n) y espera el flag de arribo de un Worker a su izquierda (módulo n) y luego lo limpia. esta clase de barrera es llamada dissemination barrier pues está basada en una técnica para diseminar información a n procesos en $\log_2 n$ rondas. En este caso, cada Worker disemina la noticia de su arribo a la barrera.

ALGORITMOS PARALELOS DE DATOS

Un *algoritmo paralelo de datos* es un algoritmo iterativo que repetidamente y en paralelo manipula un arreglo compartido. Esta clase de algoritmo está muy asociado con multiprocesadores sincrónicos, es decir máquinas SIMD. Sin embargo, son algoritmos también útiles en multiprocesadores asincrónicos.

Esta sección desarrolla soluciones paralelas a tres problemas: sumas parciales de un arreglo, encontrar el final de una lista enlazada, y etiquetado de regiones. Estos ilustran las técnicas básicas que se encuentran en los algoritmos paralelos de datos y el uso de sincronización barrier. Al final de la sección se describen los multiprocesadores SIMD y cómo remueven muchas fuentes de interferencia y por lo tanto remueven la necesidad de programar barreras.

Computación de prefijos paralela

Con frecuencia es útil aplicar una operación a todos los elementos de un arreglo. Por ejemplo, para computar el promedio de un arreglo de valores a[1:n], primero necesitamos sumar todos los elementos, y luego dividir por n. O podríamos querer saber los promedios de todos los prefijos a[1:i] del arreglo, lo cual requiere computar las sumas de todos los prefijos. A causa de la importancia de esta clase de computación, el lenguaje APL provee operadores especiales llamados reducción y scan.

En esta sección se muestra cómo computar en paralelo las sumas de todos los prefijos de un arreglo. Esto es llamado una computación *parallel prefix*. El algoritmo básico puede usarse para cualquier operador binario asociativo (suma, multiplicación, operadores lógicos, o máximo). En consecuencia, estas computaciones son útiles en muchas aplicaciones, incluyendo procesamiento de imágenes, computaciones de matrices, y parsing en un lenguaje regular.

Supongamos que tenemos un arreglo a[1:n] y queremos obtener sum[1:n], donde sum[i] es la suma de los primeros i elementos de a. La manera obvia de resolver este problema secuencialmente es iterar a través de los dos arreglos:

```
sum[1] := a[1]
fa i := 2 to n → sum[i] := sum[i-1] + a[i] af
```

En particular, cada iteración suma a[i] a la suma ya computada de los i-1 elementos anteriores.

Consideremos cómo podemos paralelizar esta aproximación. Si nuestra tarea fuera solo buscar la suma de todos los elementos, podríamos proceder como sigue. Primero, sumar pares de elementos en paralelo; por ejemplo, sumar a[1] y a[2] en paralelo con la suma de otros pares. Segundo, combinar los resultados del primer paso, nuevamente en pares; por ejemplo, sumar la suma de a[1] y a[2] con la suma de a[3] y a[4] en paralelo con la computación de otras sumas parciales. Si seguimos este proceso, en cada paso doblaríamos el número de elementos que han sido sumados. Así, en ($\log_2 n$) pasos podríamos haber computado la suma de todos los elementos. Esto es lo mejor que podemos hacer si tenemos que combinar todos los elementos de a dos a la vez.

Para computar la suma de todos los prefijos en paralelo, podemos adaptar esta técnica de doblar el número de elementos que han sido sumados. Primero, seteamos todos los sum[i] a a[i]. Luego, en paralelo sumamos sum[i-1] a sum[i], para todo i > 1. En particular, sumamos los elementos que están a distancia 1. Ahora doblamos la distancia, sumando sum[i-2] a sum[i], en este caso para todo i > 2. Si seguimos doblando la distancia, entonces después de ($\log_2 n$) rondas habremos computado todas las sumas parciales. Como ejemplo, la siguiente tabla ilustra los pasos del algoritmo para un arreglo de 6 elementos:

valores iniciales de a[1:6]	1	2	3	4	5	6
sum después de distancia 1	1	3	5	7	9	11
sum después de distancia 2	1	3	6	10	14	18
sum después de distancia 4	1	3	6	10	15	21

La siguiente es una implementación del algoritmo:

```
var a[1:n] : int, sum[1:n] : int, old[1:n] : int
Sum[i: 1..n] :: var d := 1
                sum[i] := a[i]      # inicializa los elementos de sum
                barrier
                { SUM: sum[i] = a[i-d+1] + ... + a[i] }
                do d < n →
                      old[i] := sum[i]        # salva el viejo valor
                      barrier
                      if (i-d) ≥ 1 → sum[i] := old[i-d] + sum[i] fi
                      barrier
                      d := 2 * d
                od
```

Cada proceso primero inicializa un elemento de *sum*. Luego repetidamente computa sumas parciales. En el algoritmo, **barrier** representa un punto de sincronización barrier implementado usando uno de los algoritmos ya vistos.

Se necesita que las barreras en este algoritmo eviten interferencia. Por ejemplo, todos los elementos de *sum* necesitan ser inicializados antes de que cualquier proceso los examine. Además, cada proceso necesita hacer una copia del viejo valor de *sum*[i] antes de actualizar ese valor. El invariante SUM especifica cuánto del prefijo de *a* ha sumado cada proceso en cada iteración.

Como dijimos, podemos modificar este algoritmo para usar cualquier operador binario asociativo. Todo lo que necesitamos cambiar es el operador en la sentencia que modifica *sum*. Dado que hemos escrito la expresión como old[i-d] + sum[i], el operador binario no necesita ser conmutativo. También podemos adaptar el algoritmo para usar menos de n procesos. En este caso, cada proceso sería responsable de computar las sumas parciales de una parte del arreglo.

Operaciones sobre listas enlazadas

Al trabajar con estructuras de datos enlazados tales como árboles, los programadores con frecuencia usan estructuras balanceadas tales como árboles binarios para ser capaces de buscar e insertar ítems en tiempo logarítmico. Sin embargo, usando algoritmos paralelos de datos aún muchas operaciones sobre listas lineales pueden ser implementadas en tiempo logarítmico. Aquí se muestra cómo encontrar el final de una lista enlazada serialmente. La misma clase de algoritmo puede usarse para otras operaciones sobre listas, como computar todas las sumas parciales, insertar un elemento en una lista con prioridades, o hacer matching entre elementos de dos listas.

Supongamos que tenemos una lista de hasta *n* elementos. Los links son almacenados en el arreglo *link*[1:n], y los valores de los datos en *data*[1:n]. La cabeza de la lista es apuntada por la variable *head*. Si el elemento *i* es parte de la lista, entonces *head* = *i* o *link*[j] = *i* para algún *j*, $1 \leq j \leq n$. El campo *link* del último elemento es un puntero nulo, el cual representaremos con 0. También asumimos que los campos *link* de los elementos que no están en la lista son punteros nulos y que la lista ya está inicializada.

El problema es encontrar el final de la lista. El algoritmo secuencial comienza en el elemento *head* y sigue los links hasta encontrar un enlace nulo; el último elemento visitado es el final de la lista. El tiempo de ejecución del algoritmo secuencial es proporcional a la longitud de la lista. Sin embargo, podemos encontrar el final en un tiempo proporcional al logaritmo de la longitud de la lista usando un algoritmo paralelo y la técnica de doblar introducida en la sección previa.

Asignamos un proceso *Find* a cada elemento de la lista. Sea *end*[1:n] un arreglo compartido de enteros. Si el elemento *i* es una parte de la lista, el objetivo de *Find*[i] es setear *end*[i[en el índice del final del último elemento de la lista; en otro caso *Find*[i] debería setear *end*[i] en 0. Para evitar casos especiales, asumiremos que la lista contiene al menos dos elementos.

Inicialmente cada proceso setea *end*[i] en *link*[i], es decir, al índice del próximo elemento en la lista (si lo hay). Así, *end* inicialmente reproduce el patrón de links en la lista. Luego los procesos ejecutan una serie de rondas. En cada ronda, un proceso mira *end*[*end*[i]]. Si tanto esto como *end*[i] son no nulos, entonces el proceso setea *end*[i] a *end*[*end*[i]]. Después de la primera ronda, *end*[i] apuntará a un elemento de la lista a dos links de distancia (si lo hay). Después de dos rondas, *end*[i] apuntará a un elemento a 4 links de distancia (si hay uno). Luego de $\log_2 n$ rondas, cada proceso habrá encontrado el final de la lista.

La siguiente es una implementación de este algoritmo:

```
var link[1:n] : int, end[1:n] : int
Find[i: 1..n] :: var new : int, d := 1
                 end[i] := link[i]      # inicializa los elementos de end
                 barrier
                 { FIND: end[i] = el índice del final de la lista a lo sumo 2^(d-1)
                         links de distancia del elemento i }
                 do d < n →
                       new := 0     # ve si end[i] debe ser actualizado
                       if end[i] ≠ 0 and end[end[i]] ≠ 0 → new := end[end[i]] fi
                       barrier
                       if new ≠ 0 → end[i] := new fi      # actualiza end[i]
                       barrier
                       d := 2 * d
                 od
```

Dado que la técnica de programación es la misma que en la computación de prefijos el algoritmo es estructuralmente idéntico. Nuevamente **barrier** especifica puntos de sincronización barrier, necesarios para evitar interferencia. El invariante FIND especifica a qué apunta *end*[i] antes y después de cada iteración. Si el final de la lista está a menos de 2^{d-1} links del elemento i, entonces *end*[i] no cambiará en futuras iteraciones.

Computación de grillas

Muchos problemas de procesamiento de imágenes y resolución de ecuaciones diferenciales parciales pueden resolverse usando lo que se llama *grid computations* o *mesh computations*. La idea básica es usar una matriz de puntos que superpone una grilla o red en una región espacial. En un problema de procesamiento de imágenes, la matriz es inicializada con los valores de los pixels, y el objetivo es hacer algo como encontrar conjuntos de pixels vecinos que tengan la misma intensidad. Para ecuaciones diferenciales, los bordes de la matriz son inicializados con condiciones de borde, y el objetivo es computar una aproximación para el valor de cada punto interior, lo cual corresponde a encontrar un estado que sea solución para la ecuación. En cualquier caso, el esqueleto básico de una computación de grilla es:

```
inicializar la matriz
do no terminó →
        computar un nuevo valor para cada punto
        chequear terminación
od
```

Típicamente, en cada iteración los nuevos valores de los puntos pueden computarse en paralelo.

Como ejemplo, se presenta una solución a la ecuación de Laplace en dos dimensiones: $\Delta^2(\phi) = 0$. Sea *grid*(0:n+1, 0:n+1) una matriz de puntos. Los bordes de *grid* representan los límites de una región bidimensional. Los elementos interiores de *grid* corresponden a una red que se superpone a la región. El objetivo es computar los valores de estado seguro de los puntos interiores. Para la ecuación de Laplace, podemos usar un método diferencial finito tal como la iteración de Jacobi. En particular, en cada iteración computamos un nuevo valor para cada punto interior tomando el promedio de los valores previos de sus 4 vecinos más cercanos. Este método es estacionario, por lo tanto podemos terminar la computación cuando el nuevo valor para cada punto está dentro de alguna constante EPSILON de su valor previo. El siguiente algoritmo presenta una computación de grilla que resuelve la ecuación de Laplace:

```
var grid[0:n+1, 0:n+1], newgrid[0:n+1, 0:n+1] : real
var converged : bool := false
Grid[i:1..n, j:1..n] ::
    do not converged →
        newgrid[i,j] := ( grid[i-1,j] + grid[i+1,j] + grid[i,j-1] + grid[i,j+1] ) / 4
        barrier
        chequear convergencia
        barrier
        grid[i,j] := newgrid[i,j]
        barrier
    od
```

Nuevamente usamos barreras para sincronizar los pasos de la computación. En este caso hay tres pasos por iteración: actualizar *newgrid*, chequear convergencia, y luego mover los contenidos de *newgrid* a *grid*. Se usan dos matrices para que los nuevos valores de los puntos de la grilla dependa solo de los viejos valores. La computación termina cuando los valores de *newgrid* están todos dentro de EPSILON de los de *grid*. Estas diferencias obviamente pueden chequearse en paralelo, pero los resultados necesitan ser combinados. Esto puede hacerse usando una computación de prefijos paralela.

Aunque hay un alto grado de paralelismo potencial en una computación de grilla, en la mayoría de las máquinas no puede aprovecharse totalmente. En consecuencia, es típico particionar la grilla en bloques y asignar un proceso (y procesador) a cada bloque. Cada proceso maneja su bloque de puntos; los procesos interactúan como en el algoritmo anterior.

Multiprocesadores sincrónicos

En un multiprocesador asincrónico, cada procesador ejecuta un proceso separado y los procesos ejecutan posiblemente a distintas velocidades. Los multiprocesadores asincrónicos son ejemplos de máquinas MIMD (en las cuales puede haber múltiples procesos independientes). Este es el modelo de ejecución que hemos asumido.

Aunque las máquinas MIMD son los multiprocesadores más usados y flexibles, también hay disponibles máquinas SIMD como la Connection Machene. Una SIMD tiene múltiples flujos de datos pero sólo un flujo de instrucción. En particular, cada procesador ejecuta exactamente la misma secuencia de instrucciones, y lo hacen en un lock step. Esto hace a las máquinas SIMD especialmente adecuadas para ejecutar algoritmos paralelos de datos. Por ejemplo, en una SIMD, el algoritmo para computar todas las sumas parciales de un arreglo se simplifica a:

```
var a[1:n] : int, sum[1:n] : int
Sum[i: 1..n] :: var d := 1
                sum[i] := a[i]      # inicializa los elementos de sum
                do d < n →
                    if (i-d) ≥ 1 → sum[i] := old[i-d] + sum[i] fi
                    d := 2 * d
                od
```

No necesitamos programar las barreras pues cada proceso ejecuta las mismas instrucciones al mismo tiempo; entonces cada instrucción, y por lo tanto cada referencia a memoria está seguida por una barrera implícita. Además, no necesitamos usar variables extra para mantener los viejos valores. En la asignación a *sum*[i], cada proces(ad)o(r) busca los valores viejos de *sum* antes de que cualquiera asigne nuevos valores. Por lo tanto una SIMD reduce algunas fuentes de interferencia haciendo que la evaluación de expresiones paralela aparezca como atómica.

Es tecnológicamente mucho más fácil construir una máquina SIMD con un número masivo de procesadores que construir una MIMD masivamente paralela. Esto hace a las SIMD atractivas para grandes problemas que pueden resolverse usando algoritmos paralelos de datos. Sin embargo, el desafío para el programador es mantener a los procesadores ocupados haciendo trabajo útil. En el algoritmo anterior, por ejemplo, cada vez menos procesadores actualizan *sum*[i] en cada iteración. Los procesadores para los que la guarda de la sentencia **if** es falsa simplemente se demoran hasta que los otros completen la sentencia **if**. Aunque las sentencias condicionales son necesarias en la mayoría de los algoritmos, reducen la eficiencia en las máquinas SIMD.

IMPLEMENTACION DE PROCESOS

Hemos usado la sentencia **co** y procesos y los seguiremos usando. Esta sección muestra cómo implementarlos. Primero, damos una implementación para un único procesador. Luego generalizamos la implementación para soportar la ejecución de procesos en sobre un multiprocesador con memoria compartida.

Ambas implementaciones emplean una colección de estructuras de datos y rutinas llamada *kernel* (con frecuencia llamado *núcleo*: indica que este software es común para cada procesador y es el módulo de software central). El rol del kernel es proveer un procesador virtual para cada proceso de modo que éste tenga la sensación de estar ejecutando en su propio procesador. Dado que apuntamos solo a implementar procesos, no cubrimos varios temas se sistemas operativos que ocurren en la práctica, como alocación dinámica, scheduling de prioridades, memoria virtual, control de dispositivos, acceso a archivos, o protección.

Recordemos que los procesos son sólo abreviaciones de las sentencias **co**. Así, es suficiente mostrar cómo implementar las sentencias **co**. Consideremos el siguiente fragmento de programa:

(3.26) **var** variables compartidas
S_0
co $P_1 : S_1$ // ..... // $P_n : S_n$ **oc**
S_{n+1}

Los P_i son nombres de procesos. Los S_i son listas de sentencias y declaraciones opcionales de variables locales del proceso. Necesitamos tres mecanismos diferentes para implementar (3.26):

- uno para crear procesos y comenzar a ejecutarlos
- uno para detener un proceso
- un tercero para determinar que la sentencia **co** fue completada

Una *primitiva* es una rutina que es implementada por un kernel de manera tal que aparece como una instrucción atómica. Crearemos y destruiremos procesos con dos primitivas del kernel: **fork** y **quit**. Cuando un proceso invoca **fork**, es creado otro proceso y se hace elegible para ejecución. Los argumentos de **fork** indican las direcciones de la primera instrucción a ser ejecutada por el nuevo proceso y cualquier otro dato necesario para especificar su estado inicial (por ejemplo, parámetros). Cuando un proceso invoca **quit**, deja de existir.

Un kernel usualmente provee una tercera primitiva para permitirle a un proceso demorarse hasta que otro proceso termina. Sin embargo, dado que en este capítulo los procesos se sincronizan por medio de busy waiting, detectaremos terminación usando un arreglo global de variables booleanas. En particular, podemos usar estas variables globales, **fork** y **quit** para implementar (3.26) como sigue:

(3.27) **var** done[1:n] : **bool** := ([n] false), otras variables compartidas
S_0
crear los procesos, luego esperar que terminen
fa i := 1 **to** n → **fork**(P_i) **af**
fa i := 1 **to** n → **do not** done[i] → **skip od af**
S_{n+1}

Cada uno de los P_i ejecuta el siguiente código:

P_i : S_i ; done[i] := true; **quit**( )

Asumimos que el proceso principal en (3.27) es creado implícitamente de modo tal que automáticamente comienza su ejecución. También asumimos que el código y los datos para todos los procesos ya están almacenados en memoria cuando comienza el proceso principal.

Se presenta un kernel monoprocesador que implementa **fork** y **quit**. También se describe cómo hacer el scheduling de procesos de modo de que cada uno tenga una chance periódica de ejecutar.

Un Kernel Monoprocesador

Todo kernel contiene estructuras de datos que representan procesos y tres tipos básicos de rutinas: manejadores de interrupción, las primitivas en sí mismas, y un dispatcher. El kernel puede contener otras estructuras de datos y funcionalidad, como descriptores de archivos y rutinas de acceso a archivos. Enfocamos sólo las partes de un kernel que implementan procesos.

Hay tres maneras básicas de organizar un kernel:

- como una unidad monolítica en la cual cada primitiva se ejecuta como una acción atómica
- como una colección de procesos especializados que interactúan para implementar primitivas del kernel (por ejemplo, uno puede manejar la E/S de archivos y otro el manejo de memoria), o
- como un programa concurrente en el cual más de un proceso usuario puede estar ejecutando una primitiva del kernel al mismo tiempo

Usaremos aquí la primera aproximación pues es la más simple para un kernel monoprocesador pequeño. Usaremos la tercera en un kernel multiprocesador.

Cada proceso es representado en el kernel por un *descriptor de proceso*. Cuando un proceso está ocioso, su descriptor contiene toda la información necesaria para ejecutar el proceso. Esto incluye la dirección de la próxima instrucción que ejecutará el proceso y los contenidos de los registros del procesador.

El kernel comienza a ejecutar cuando ocurre una interrupción. Las interrupciones pueden ser divididas en dos grandes categorías: interrupciones externas de los dispositivos periféricos e interrupciones internas de los procesos ejecutantes. Cuando ocurre una interrupción, el procesador automáticamente salva la información de estado suficiente para que el proceso interrumpido pueda ser reanudado. Luego el procesador entra un *interrupt handler*; normalmente hay un manejador para cada clase de interrupción.

Para invocar una primitiva del kernel, un proceso causa una interrupción interna ejecutando una instrucción de máquina normalmente llamada supervisor call (SVC) o trap. El proceso pasa un argumento con la instrucción SVC para indicar qué primitiva debe ejecutarse y pasa otros argumentos en registros. El manejador de interrupción del SVC primero salva el estado completo del proceso ejecutante. Luego llama a la primitiva apropiada, la cual es implementada dentro del kernel por un procedure. Cuando se completa la primitiva, se llama al *dispatcher* (scheduler del procesador). El dispatcher selecciona un proceso para su ejecución y luego carga su estado.

Para asegurar que las primitivas son ejecutadas atómicamente, la primera acción de un interrupt handler es inhibir otras interrupciones; la última acción del dispatcher es habilitar las interrupciones. Cuando ocurre una interrupción, las futuras interrupciones son inhibidas automáticamente por el hardware; el kernel rehabilita las interrupciones como un efecto lateral de cargar un estado de proceso. Podemos ver las componentes del kernel y el flujo de control a través del kernel de la siguiente manera:

interrupciones externas ⇒
interrupciones internas ⇒
manejadores ⇒ primitivas ⇒ dispatcher ⇒

El control fluye en una dirección desde los manejadores a través de las primitivas hasta el dispatcher y luego nuevamente a un proceso activo.

Representaremos los descriptores de proceso por un arreglo:

var *process_descriptor*[1:n] : *process_type*

El *process_type* es un registro que describe los campos de un descriptor. Cuando se le pide al kernel crear un nuevo proceso, aloca e inicializa un descriptor vacío. Cuando el dispatcher del kernel schedules un proceso, necesita encontrar el descriptor de un proceso que es elegible para ejecutar. Ambas funciones podrían ser implementadas buscando a través del arreglo *process_descriptor*, asumiendo que cada registro contiene un campo indicando si el entry está libre o en uso. Sin embargo, normalmente se mantienen dos listas: una *free list* de descriptores vacíos y una *ready list* de descriptores de procesos que están esperando turno para ejecutarse. Usaremos esta representación. Hay una variable adicional, *executing*, que contiene el índice del descriptor del proceso que está ejecutando.

Con esta representación para las estructuras de datos del kernel, la primitiva *fork* toma un descriptor de la free list, lo inicializa y lo inserta al final de la ready list. La primitiva *quit* pone el descriptor del proceso ejecutante en la free list y setea *executing* en 0 para indicarle al dispatcher que el proceso ya no quiere ejecutar.

Cuando el dispatcher es llamado al final de una primitiva, chequea el valor de *executing*. Si es 0, remueve el primer descriptor de la ready list y setea *executing* para que apunte a él. (Si *executing* no es 0. el proceso que está ejecutando continúa ejecutando). Luego el dispatcher carga el estado del proceso que ejecutará a continuación. Asumimos que la ready list es una cola FIFO.

Un tema que falta es asegurar fairness en la ejecución de procesos. Si el proceso ejecutante siempre terminara en un tiempo finito. la implementación anterior del kernel aseguraría fairness pues asume que la ready list es una cola FIFO. Sin embargo, si algún proceso (tal como el proceso principal en (3.27)) espera una condición que no será nunca verdadera, se quedará dando vueltas para siempre excepto que sea forzado a ceder el procesador. Podemos usar un interval timer para asegurar que los procesos liberan periódicamente el control del procesador, asumiendo que tal timer está provisto por hardware. Esto, más el hecho de que la ready list es una cola FIFO, garantiza que cada proceso tiene una chance periódica de ejecutar.

Un *interval timer* es un dispositivo que, cuando es inicializado con un valor entero positivo, decrementa el valor a una velocidad fija y activa una interrupción de timer cuando el valor llega a 0. El kernel usa tal timer como sigue: Primero, antes de cargar el estado del proceso que ejecutará a continuación, el dispatcher inicializa el timer. Luego, si ocurre una interrupción de timer, el manejador ubica el descriptor de *executing* al final de la ready list, setea *executing* a 0, y luego llama al dispatcher. Esto causa que los procesos tengan un turno para ejecutar de manera round-robin.

Juntando todo esto, tenemos el siguiente esqueleto de kernel monoprocesador:

```
var process_descriptors[1:n] : process_type
var executing : int
var variables para representar free list y ready list
SVC_Handler:      # se entra con las interrupciones automáticamente inhibidas
    salva el estado de executing
    determina qué primitiva fue invocada, luego la llama

Timer_Handler:    # se entra con las interrupciones automáticamente inhibidas
    inserta el descriptor de executing al final de la ready list; executing := 0
    call dispatcher( )

procedure fork ( estado inicial del proceso )
    remover un descriptor de la free list
    inicializar el descriptor
    insertar el descriptor al final de la ready list
    call dispatcher( )
end

procedure quit ( )
    insertar el descriptor de executing al final de la free list; executing := 0
    call dispatcher( )
end

procedure dispatcher ( )
    if executing = 0 → remover descriptor del inicio de la ready list
                       setear executing para que apunte a él
    fi
    comenzar el interval timer
    cargar el estado de executing      # interrupciones automáticamente
habilitadas
end
```

Asumimos que cuando el kernel es inicializado (lo que sucede como resultado de "bootear" el procesador) se crea un proceso y se hace proceso ejecutante. También asumimos que un efecto lateral de comenzar un interval timer en dispatcher es deshabilitar cualquier interrupción que podría haber estado pendiente como consecuencia de que el timer alcanzó 0 mientras se está ejecutando. Con esta suposición, el proceso que es seleccionado para ejecución no será interrumpido inmediatamente perdiendo el control del procesador.

Este kernel ignora excepciones que podrían ocurrir, tales como que la free list esté vacía cuando se llama a fork. Esta implementación también asume que siempre hay al menos un proceso listo. En la práctica, un kernel podría tener siempre un proceso "do nothing" que es despachado cuando no hay trabajo útil para hacer.

Un Kernel Multiprocesador

Un multiprocesador de memoria compartida tiene dos o más procesadores y al menos algo de memoria que es accesible por cualquier procesador. Es relativamente directo extender el kernel monoprocesador a multiprocesador. Los cambios principales que necesitamos hacer son almacenar los procedimientos y estructuras de datos del kernel en memoria compartida, acceder las estructuras de datos con exclusión mutua cuando sea necesario para evitar interferencia, y cambiar el dispatcher para explotar los múltiples procesadores. Sin embargo hay algunas sutilezas, que resultan de las características especiales de los multiprocesadores.

Asumimos que las interrupciones internas (traps) son servidas por el procesador que estaba ejecutando el proceso que causó el trap y asumimos que cada procesador tiene un interval timer. Por ahora, también asumimos que cada operación del kernel y cada proceso pueden ser ejecutados en cualquier procesador.

Cuando un procesador es interrumpido, entra al kernel e inhibe las interrupciones en ese procesador. Esto hace la ejecución dentro del kernel indivisible en ese procesador, pero no previene a otros procesadores de ejecutar simultáneamente en el kernel. Para prevenir interferencia entre procesadores, podríamos hacer al kernel entero una SC. Pero esto es malo por dos razones. Primero, impide innecesariamente alguna ejecución concurrente segura. En particular, solo el acceso a estructuras de datos compartidas como las listas free y ready es crítico. Segundo, hacer al kernel entero una SC resulta en secciones críticas innecesariamente largas. Esto disminuye la performance pues demora a los procesadores tratando de entrar al kernel, e incrementa memory contention para las variables que implementan el protocolo de SC del kernel. El siguiente principio da una mejor elección:

(3.28) **Principio de Lockeo de Multiprocesador**. Hacer secciones críticas cortas protegiendo individualmente cada estructura de datos crítica. Usar SC separadas (con variables separadas para los protocolos de entrada y salida) para cada estructura de datos crítica.

En nuestro kernel, los datos críticos son las listas free y ready. Para proteger el acceso a ellas, podemos usar cualquiera de los protocolos de SC vistos. En un multiprocesador particular, la elección de cual usar estará afectada por las instrucciones especiales de que se dispone.

Dado que asumimos que los traps son manejados por el procesador en que ocurren y que cada procesador tiene su propio interval timer, los manejadores de trap y timer son esencialmente los mismos que en el kernel monoprocesador. Las dos diferencias son que *executing* es un arreglo, con una entrada por procesador, y que Timer_Handler necesita lockear y liberar la lista entera.

El código para las tres primitivas del kernel son esencialmente las mismas. Las diferencias son las ya marcadas.

Los mayores cambios están en el código del dispatcher. Antes teníamos un procesador y asumíamos que siempre tenía un proceso para ejecutar. Ahora podría haber menos procesos que procesadores, con lo cual habría procesadores ociosos. Cuando un nuevo proceso es forked (o despertado luego de una interrupción de E/S) necesita ser asignado a un procesador ocioso, si lo hay. Esta funcionalidad puede proveerse de tres maneras distintas:

- Que cada procesador, cuando está ocioso, ejecute un proceso especial que periódicamente examina la ready list hasta que encuentra un proceso listo.
- Tener un procesador ejecutando búsqueda de fork para un procesador ocioso y le asigne el nuevo proceso.
- Usar un proceso dispatcher separado que ejecute en su propio procesador y continuamente intente asignar procesos listos a procesadores ociosos.

Dado que los procesadores ociosos no tienen nada que hacer hasta que encuentran algún proceso para ejecutar, usaremos la primera aproximación. En particular, cuando el dispatcher encuentra que la ready list está vacía, setea *executing*[i] para que apunte al descriptor de un proceso ocioso. Ese proceso en el procesador *i* ejecuta el siguiente código:

Idle :: **do** executing[i] = proceso Idle $\rightarrow$

```
        do ready list vacía → Delay od
        lockear la ready list
        if ready list no vacía →      # se necesita chequear nuevamente
            remover descriptor del inicio de la ready list
            setear executing[i] para que apunte a él
        fi
        liberar la ready list
    od
    comenzar el interval timer en el procesador i
    cargar el estado de executing[i]        # con interrupciones habilitadas
```

En esencia, Idle es un self-dispatcher. Primero da vueltas hasta que la ready list no esté vacía, luego remueve un descriptor y comienza a ejecutar ese proceso. Para evitar memory contention, Idle no debería examinar continuamente la ready list o continuamente lockearla y liberarla. Así, usamos un protocolo test-and-test-and-set similar al mostrado en (3.7) y (3.9). Dado que la ready list podría estar vacía antes de que Idle adquiera el lock de ella, necesita volver a testear.

Lo que resta es asegurar fairness. Nuevamente, emplearemos timers para asegurar que los procesos que ejecutan fuera del kernel son forzados a ceder los procesadores. Asumimos que cada procesador tiene su propio timer, que es usado como en el kernel monoprocesador. Sin embargo, solamente los timers no son suficientes pues los procesos ahora pueden ser demorados dentro del kernel esperando adquirir acceso a las estructuras de datos compartidas del kernel. Así, necesitamos usar una solución fair al problema de la SC tal como los algoritmos tie-breaker, ticket o bakery. Si usamos el protocolo test-and-set, hay posibilidad de que los procesos sufran inanición (aunque esto es poco probable pues las SC en el kernel son muy cortas).

El siguiente es un esqueleto de kernel multiprocesador que incorpora estas suposiciones y decisiones. La variable *i* es el índice del procesador que está ejecutando las rutinas, y lockear y liberar son protocolos de entrada y salida a SC. Nuevamente, se ignoran posibles excepciones y no se incluye código para los manejadores de interrupción de E/S, manejo de memoria, etc.

```
var process_descriptors[1:n] : process_type
var executing[1:p] : int      # un entry por procesador
var variables para representar free list y ready list y sus locks
SVC_Handler:      #  entra con interrupciones automáticamente inhibidas en
proc. i
    salva el estado de executing[i]
    determina qué primitiva fue invocada, luego la llama

Timer_Handler:  # entra con interrupciones automáticamente inhibidas en proc.
i
    lockear ready list; insertar executing[i] al final; liberar ready list
    executing[i] := 0
    call dispatcher( )

procedure fork ( estado inicial del proceso )
    lockear free list; remover un descriptor; liberar free list
    inicializar el descriptor
    lockear ready list; insertar descriptor al final; liberar ready list
    call dispatcher( )
end

procedure quit ( )
    lockear free list; insertar executing[i] al final; liberar free list
    executing[i] := 0
    call dispatcher( )
end

procedure dispatcher ( )
    if executing[i] = 0 →
        lockear ready list
        if ready list no vacía → remover descriptor de ready list
```

```
                                    setear executing[i] para que apunte a él
                □ ready list vacía → setear executing[i] apuntando al proceso Idle
                fi
                liberar ready list
            fi
            if executing[i] no es el proceso Idle → comenzar timer en el procesador i fi
            cargar el estado de executing[i]      # interrupciones automáticamente
        habilitadas
        end
```

Este kernel emplea una sola ready list que se asume una cola FIFO. Si los procesos tienen distintas prioridades, la lista necesita ser una cola con prioridades. Pero esto causaría que la ready list se convierta en un cuello de botella. Una posible solución es usar un conjunto de colas.

Semáforos

Los protocolos de sincronización que usan solo busy waiting pueden ser difíciles de diseñar, entender y probar su corrección. La mayoría de estos protocolos son bastante complejos, no hay clara separación entre las variables usadas para sincronización y las usadas para computar resultados. Una consecuencia de estos atributos es que se debe ser muy cuidadoso para asegurar que los procesos se sincronizan correctamente.

Otra deficiencia es que es ineficiente cuando los procesos son implementados por multiprogramación. Un procesador que está ejecutando un proceso “spinning” podría ser empleado más productivamente por otro proceso. Esto también ocurre en un multiprocesador pues usualmente hay más procesos que procesadores.

Dado que la sincronización es fundamental en los programas concurrentes, es deseable tener herramientas especiales que ayuden en el diseño de protocolos de sincronización correctos y que puedan ser usadas para bloquear procesos que deben ser demorados. Los *Semáforos* son una de las primeras de tales herramientas y una de las más importantes. Hacen fácil proteger SC y pueden usarse de manera disciplinada para implementar sincronización por condición. Los semáforos pueden ser implementados de más de una manera. En particular, pueden implementarse con busy waiting, pero también interactuando con un proceso scheduler para obtener sincronización sin busy waiting.

El concepto de semáforo es motivado por una de las maneras en que el tráfico de trenes es sincronizado para evitar colisiones: es una flag de señalización que indica si la pista está desocupada o hay otro tren. Los semáforos en los programas concurrentes proveen un mecanismo de señalización básico y se usan para implementar exclusión mutua y sincronización por condición.

NOTACION Y SEMANTICA

Un semáforo es una instancia de un tipo de datos abstracto: tiene una representación que es manipulada solo por dos operaciones especiales: **P** y **V**. La operación **V** señala la ocurrencia de un evento; la operación **P** se usa para demorar un proceso hasta que ocurra un evento. En particular, las dos operaciones deben ser implementadas de modo de preservar la siguiente propiedad para todo semáforo en un programa:

(4.1) **Invariante de Semáforo**. Para el semáforo *s*, sea *nP* el número de operaciones **P** completadas, y *nV* el número de operaciones **V** completadas. Si *init* es el valor inicial de *s*, entonces en todos los estados de programa visibles, $nP \leq nV + init$.

Así, la ejecución de una operación **P** potencialmente demora hasta que se hayan ejecutado un número adecuado de operaciones **V**.

La manera más simple de proveer la sincronización requerida es representar cada semáforo por un entero no negativo *s* que registra el valor inicial más la diferencia entre el número de operaciones **V** y **P** completadas; en particular, $s = init + nV - nP$. Con esta representación, el invariante del semáforo se convierte en:

SEM: $s \geq 0$

Dado que una operación **P** exitosa incrementa implícitamente *nP*, decrementa *s*. Para que SEM sea un invariante global, el decremento debe estar guardado pues:

$wp(s := s-1, s \geq 0) = s-1 \geq 0 = s > 0$

Una operación **V** incrementa implícitamente *nV* con lo cual incrementa *s*. Este incremento no necesita estar guardado pues:

$s \geq 0 \Rightarrow wp(s := s+1, s \geq 0)$

Estas observaciones llevan a las siguientes definiciones de **P** y **V**:

P(s): ⟨ **await** s > 0 → s := s - 1 ⟩
V(s): ⟨ s := s + 1 ⟩

Ambas operaciones son acciones atómicas. También son las *únicas* operaciones sobre semáforos, es decir: el valor no puede ser examinado directamente.

Los semáforos así definidos se llaman *semáforos generales*: el valor de *s* puede ser cualquier entero no negativo. Un *semáforo binario* es un semáforo cuyo valor es solo 0 o 1. En particular, un semáforo binario *b* satisface un invariante global más fuerte:

BSEM: $0 \leq b \leq 1$

Mantener este invariante requiere guardar la operación **V** (o causar que el programa aborte si la operación **V** es ejecutada cuando *b* es 1). Así, las operaciones sobre un semáforo binario tienen las siguientes definiciones:

P(b): ⟨ **await** b > 0 → b := b - 1 ⟩
V(b): ⟨ **await** b < 1 → b := b + 1 ⟩

Siempre que un semáforo binario se usa de tal manera que una operación **V** es ejecutada solo cuando b es 0, **V**(b) no causará demora (Con frecuencia esto se asume y por lo tanto **V**(b) se define simplemente como ⟨ b := b + 1 ⟩). Sin embargo, esta definición es incorrecta si **V** es ejecutada inadvertidamente cuando *b* es 1 pues entonces BSEM no será invariante.

Declararemos semáforos en los programas usando un tipo especial **sem**. El valor inicial por defecto de cada semáforo es 0. Cuando queremos un valor inicial diferente usaremos una asignación en la declaración:

var *mutex* : **sem** := 1

Esto es lo mismo que ejecutar implícitamente el número apropiado de operaciones **V**. Para un semáforo general, el valor inicial debe ser no negativo; para un semáforo binario, debe ser 0 o 1. Podemos usar arreglos de semáforos:

var *forks*[1:5] : **sem** := ([5] 1)

Dado que las operaciones sobre semáforos son definidas en términos de sentencias **await**, su semántica formal sigue directamente de las aplicaciones de la Regla de Sincronización (2.7). Recordemos esa regla de inferencia:

Regla de Sincronización:
$$\frac{\{ P \wedge B \}\ S\ \{ Q \}}{\{ P \}\ \langle\ \textbf{await}\ B \rightarrow S\ \rangle\ \{ Q \}}$$

Sea *g* un semáforo general. Sustituyendo *g* > 0 por B, y *g* := *g* - 1 por S, la hipótesis se convierte en:

$\{ P \wedge g > 0 \}\ g := g - 1\ \{ Q \}$

Usando el Axioma de Asignación, esto se simplifica a:

$(P \wedge g > 0) \Rightarrow Q^{g}_{g-1}$

La regla de inferencia para **V**(g) es similar: sustituir *true* por B y *g* := *g* + 1 por S en la Regla de Sincronización, y simplificar las hipótesis.

Haciendo las mismas sustituciones en la conclusión de la Regla de Sincronización, tenemos:

Regla de Semáforos Generales:

$$\frac{\{ P \wedge g > 0 \} \Rightarrow Q^{g}_{g-1}}{\{ P \}\ \mathbf{P}(g)\ \{ Q \}}$$

$$\frac{P \Rightarrow Q^{g}_{g+1}}{\{ P \}\ \mathbf{V}(g)\ \{ Q \}}$$

Las reglas para semáforos binarios son casi idénticas a las de los generales. La única diferencia resulta del hecho de que **V**(b) es una acción atómica condicional, y por lo tanto las hipótesis incluyen la condición b < 1 en el antecedente de la implicación:

Regla de Semáforos Binarios:

$$\frac{\{ P \wedge b > 0 \} \Rightarrow Q^{b}_{b-1}}{\{ P \}\ \mathbf{P}(b)\ \{ Q \}}$$

$$\frac{\{ P \wedge b < 1 \} \Rightarrow Q^{b}_{b+1}}{\{ P \}\ \mathbf{V}(b)\ \{ Q \}}$$

Los atributos de fairness de las operaciones sobre semáforos también se desprenden del hecho que son definidas en términos de sentencias **await**. Sea *s* un semáforo binario o general. Usando la terminología del capítulo 2, si *s* > 0 se vuelve true y se mantiene true, la ejecución de **P**(s) terminará si la política de scheduling subyacente es débilmente fair. Si *s* > 0 es true con infinita frecuencia, la ejecución de **P**(s) terminará si la política es fuertemente fair. La operación **V** sobre un semáforo binario tiene atributos de fairness similares. Dado que la operación **V** sobre un semáforo general es una acción atómica incondicional, terminará si la política de scheduling es incondicionalmente fair. Como veremos, podemos implementar semáforos de modo que los procesos demorados sean despertados en el orden en que fueron demorados. En este caso, un proceso demorado en una operación **P** será capaz de seguir si otros procesos ejecutan un número adecuado de operaciones **V**.

USOS BASICOS Y TECNICAS DE PROGRAMACION

Los semáforos soportan directamente la implementación de protocolos de SC. También soportan directamente formas simples de sincronización por condición en las cuales las condiciones representan la ocurrencia de eventos. Esta sección ilustra estos usos derivando soluciones a cuatro problemas: secciones críticas, barreras, productores/consumidores, y buffers limitados. Las soluciones ilustran maneras adicionales de especificar propiedades de sincronización. También ilustran dos técnicas de programación importantes: cambiar variables y dividir semáforos binarios. Secciones posteriores muestran cómo usar estas técnicas para construir soluciones a problemas de sincronización más complejos.

Secciones críticas: Cambio de variables

Recordemos que, en el problema de la SC, cada uno de los *n* procesos P[1:n] ejecutan repetidamente una SC de código, en la cual se requiere acceso exclusivo a algún recurso compartido, y luego una SNC, en la cual se usan solo objetos locales. Un invariante global especificando la propiedad de exclusión mutua fue especificado de distintas maneras en el capítulo 3. Dado que las operaciones sobre semáforos manipulan enteros, cuando son usadas para sincronización, siempre es mejor especificar propiedades invariantes usando enteros (pues las acciones atómicas pueden convertirse en operaciones sobre semáforos).

Sea *in*[i] 1 cuando P[i] está en su SC, y 0 en otro caso. La propiedad requerida es que a lo sumo un proceso a la vez esté en su SC. Podemos especificar esto directamente con el predicado:

CS: in[1] + ... + in[n] ≤ 1

Alternativamente, podemos especificar el estado malo en el cual más de un proceso está en su SC por:

BAD: in[1] + ... + in[n] > 1

Dado que todos los in[i] son 0 o 1, CS = ¬BAD, por lo tanto las especificaciones son las mismas.

Dada esta especificación, tenemos el siguiente esqueleto de solución:

```
var in[1:n] : int := ( [n] 0 )
{ CS: in[1] + ..... + in[n] ≤ 1 }
P[i: 1..n] :: do true → { in[i] = 0 }
                        in[i] := 1
                        { in[i] = 1 }
                        critical section
                        in[i] := 0
                        { in[i] = 0 }
                        non-critical section
              od
```

Los procesos comparten el arreglo in[1:n], con cada proceso seteando y limpiando su elemento de *in* antes y después de ejecutar su SC. Inicialmente todos los elementos de *in* son cero, de modo que el predicado CS es inicialmente true. Cada proceso contiene aserciones acerca del elemento de in manipulado por ese proceso. Estas aserciones se desprenden de las acciones del proceso: no son interferidas pues cada proceso manipula un elemento diferente de *in*. Así, el esqueleto de solución es una proof outline válida. Pero no es lo suficientemente fuerte para concluir que la ejecución de las SC son mutuamente exclusivas. Para esto, debemos incluir el invariante CS en la proof outline.

El tercer paso de derivación es guardar las asignaciones para asegurar que CS es true luego de cada acción atómica. Consideremos la primera asignación, la cual setea *in*[i] y así establece *in*[i] = 1. Computar la precondición weakest nos da:

wp(in[i] := 1, in[i] = 1 ∧ CS) = (1 + in[1] + ... + in[n] ≤ 1)

Dado que todos los elementos de *in* son 0 o 1, esto se simplifica a:

in[1] + ... + in[n] = 0

Usamos esto para guardar la primera acción atómica en cada proceso. Para las asignaciones que limpian in[i], computando *wp* tenemos:

wp(in[i] := 0, in[i] = 0 ∧ CS) = (in[1] + ... + in[n] ≤ 1)

Dado que la precondición in[i] = 1 ∧ CS implica que esto es true, no necesitamos guardar la segunda acción atómica. Agregando la guarda a la primera acción atómica, tenemos la solución coarse grained:

```
var in[1:n] : int := ( [n] 0 )
{ CS: in[1] + ..... + in[n] ≤ 1 }
P[i: 1..n] :: do true → { in[i] = 0 ∧ CS }
                        ⟨ await in[1] + ... + in[n] = 0 → in[i] := 1 ⟩
                        { in[i] = 1 ∧ CS }
                        critical section
                        in[i] := 0
                        { in[i] = 0 ∧ CS }
                        non-critical section
              od
```

Dado que la derivación de esta solución aseguró que CS es invariante, la proof outline es válida. Así podemos usar esto y el método de Exclusión de Configuraciones (2.25) para probar que la solución asegura exclusión mutua, está libre de deadlock, y evita demora innecesaria. Por ejemplo, la exclusión mutua se desprende de:

$$(in[i] = 1 \wedge in[j] = 1 \wedge i \neq j \wedge CS) = false$$

Si el scheduling es fuertemente fair, la solución también asegura entrada eventual a una SC.

El paso de derivación que falta es usar semáforos para implementar las acciones atómicas. Podemos hacer esto *cambiando variables* para que cada sentencia atómica se convierta en una operación de semáforo. Sea *mutex* un semáforo cuyo valor es:

mutex = 1 - (in[1] + ... + in[n])

Usamos este cambio de variables pues hace a *mutex* un entero no negativo, como se requiere para un semáforo. Con este cambio, podemos reemplazar la primera acción atómica en la solución coarse grained por:

⟨ **await** mutex > 0 → mutex := mutex - 1; in[i] := 1 ⟩

Y podemos reemplazar la última acción atómica por:

⟨ mutex := mutex + 1; in[i] := 0 ⟩

Luego de hacer estos cambios, *in* se convierte en una variable auxiliar (es usada solo en asignaciones a sí misma). Así el programa tiene las mismas propiedades luego de usar la Regla de Variable Auxiliar (2.22) para borrar las variables auxiliares. Lo que se dejan son sentencias atómicas que son simplemente las operaciones **P** y **V** sobre el semáforo *mutex*. Así, tenemos la solución final:

```
var mutex : sem := 1
P[i: 1..n] :: do true → P(mutex)
                        critical section
                        V(mutex)
                        non-critical section
              od
```

Esta solución funciona para cualquier número de procesos, y es mucho más simple que las soluciones busy-waiting del capítulo 3.

La técnica de cambiar variables que usamos lleva a una solución compacta. Esto es porque permite que las sentencias **await** sean implementadas directamente por operaciones semáforo. De hecho, la técnica puede ser usada siempre que las siguientes condiciones son verdaderas:

(4.2) **Condiciones para cambiar variables**. Las acciones atómicas pueden ser implementadas por operaciones semáforo si se dan las siguientes condiciones:

(a) Distintas guardas referencian conjuntos disjuntos de variables, y estas variables son referenciadas solo en sentencias atómicas.
(b) Cada guarda puede ponerse en la forma *expr* > 0, donde *expr* es una expresión entera.
(c) Cada sentencia atómica guardada contiene una asignación que decrementa el valor de la expresión en la guarda transformada.
(d) Cada sentencia atómica no guardada incrementa el valor de la expresión en una guarda transformada.

Cuando se dan estas condiciones, se usa un semáforo para cada guarda diferente. Las variables que estaban en las guardas luego se convierten en variables auxiliares, y las sentencias atómicas se simplifican a operaciones semáforos.

Barreras: Señalizando eventos

Introdujimos la sincronización barrier en el capítulo 3 como medio para sincronizar etapas de algoritmos iterativos paralelos. Las implementaciones busy-waiting usaban variables flag que los procesos seteaban y limpiaban al llegar y dejar una barrera. Esta sección muestra cómo implementar barreras usando semáforos. La idea básica es usar un semáforo para cada flag de sincronización. Un proceso setea un flag ejecutando una operación **V**; un proceso espera a que un flag sea seteado y luego lo limpia ejecutando una operación **P**.

Consideremos primero el problema de implementar una barrera para dos procesos. Recordemos que se requieren dos propiedades. Primero, ningún proceso puede pasar la barrera hasta que ambos hayan arribado. Segundo, la barrera debe ser reusable pues en general los mismos procesos necesitarán sincronizar luego de cada etapa de la computación. Para el problema de la SC, necesitamos solo una variable por proceso para especificar la sincronización requerida pues lo único que interesaba era si un proceso estaba dentro o fuera de su SC. Aquí necesitamos saber cada vez que un proceso llega o parte de la barrera, y entonces necesitamos relacionar los estados de los dos procesos.

La manera de especificar sincronización barrier (y clases similares de sincronización) es usar contadores incrementales. Cada contador registra cuando un proceso alcanza un punto de ejecución crítico. Sea que *arrive1* y *depart1* cuentan el número de veces que el proceso P1 llega o parte de la barrera, y que *arrive2* y *depart2* juegan un rol similar para el proceso P2. Los cuatro contadores son inicialmente 0. Entonces la sincronización barrier es especificada por el predicado:

BARRIER: depart1 $\leq$ arrive2 $\wedge$ depart2 $\leq$ arrive1

Esto dice que P1 no puede pasar la barrera más veces de las que P2 llegó y, simétricamente, que P2 no puede pasar más veces que las que llegó P1. La solución y proof outline es la siguiente:

```
var arrive1 := 0, depart1 := 0, arrive2 := 0, depart2 := 0
{ BARRIER: depart1 ≤ arrive2 ∧ depart2 ≤ arrive1 }
P1:: do true → { arrive1 = depart1 }
                ⟨ arrive1 := arrive1 + 1 ⟩
                { arrive1 = depart1 + 1 }
                ⟨ depart1 := depart1 + 1 ⟩
                { arrive1 = depart1 }
     od
P2:: do true → { arrive2 = depart2 }
                ⟨ arrive2 := arrive2 + 1 ⟩
                { arrive2 = depart2 + 1 }
                ⟨ depart2 := depart2 + 1 ⟩
                { arrive2 = depart2 }
     od
```

El próximo paso es asegurar que BARRIER es un invariante global. No necesitamos guardar las asignaciones a arrive1 o arrive2, pero sí las asignaciones a depart1 y depart2 pues estos valores necesitan ser limitados. Como es usual, podríamos usar *wp* para calcular la guarda. Sin embargo, por inspección es claro que depart1 debe ser menor que arrive2 antes de ser incrementada, y similarmente depart2 debe ser menor que arrive1. Agregando las guardas obtenemos la siguiente solución y proof outline:

```
var arrive1 := 0, depart1 := 0, arrive2 := 0, depart2 := 0
{ BARRIER: depart1 ≤ arrive2 ∧ depart2 ≤ arrive1 }
P1:: do true → { BARRIER ∧ arrive1 = depart1 }
                ⟨ arrive1 := arrive1 + 1 ⟩
                { BARRIER ∧ arrive1 = depart1 + 1 }
                ⟨ await depart1 < arrive2 → depart1 := depart1 + 1 ⟩
                { BARRIER ∧ arrive1 = depart1 }
     od
```

```
P2:: do true → { BARRIER ∧ arrive2 = depart2 }
                ⟨ arrive2 := arrive2 + 1 ⟩
                { BARRIER ∧ arrive2 = depart2 + 1 }
                ⟨ await depart2 < arrive1 → depart2 := depart2 + 1 ⟩
                { BARRIER ∧ arrive2 = depart2 }
   od
```

Como es usual, el paso final es implementar las sentencias **await**. Nuevamente podemos usar la técnica de cambiar variables pues las condiciones requeridas (4.2) se cumplen: las guardas son disjuntas, cada una puede ponerse de la forma *expr* > 0, las sentencias guardadas decrementan la expresión en su guarda, y las sentencias no guardadas incrementan la expresión en alguna guarda. En particular, introducimos dos nuevas variables:

```
barrier1 = arrive1 - depart2
barrier2 = arrive2 - depart1
```

Estas variables son inicialmente 0. Cambiamos las guardas en las sentencias **await** de la solución anterior para usar estas variables, y agregamos asignaciones a ellas a las acciones atómicas que cambian los contadores originales. Por ejemplo, reemplazamos el incremento de arrive1 en P1 por:

```
⟨ barrier1 := barrier1 + 1; arrive1 := arrive1 + 1 ⟩
```

Y reemplazamos la sentencia await guardada en P1 por:

```
⟨ await barrier2 > 0 → barrier2 := barrier2 -1; depart1 := depart1 + 1 ⟩
```

Después de hacer este cambio de variables, las cuatro variables originales se convierten en variables auxiliares. Borrando las variables auxiliares, las acciones sobre barrier1 y barrier2 son simplemente operaciones sobre semáforos. Luego, tenemos la siguiente solución final:

```
var barrier1 : sem := 0, barrier2 : sem := 0
P1 :: do true → V(barrier1)
                P(barrier2)
   od
P2 :: do true → V(barrier2)
                P(barrier1)
   od
```

En esta solución, los semáforos se usan como *señales* que indican cuando ocurren eventos. Cuando un proceso llega a la barrera, señala ese evento ejecutando una operación **V** sobre un semáforo. El otro proceso espera el evento ejecutando una operación **P** sobre el mismo semáforo. Dado que la sincronización barrier es simétrica, cada proceso toma las mismas acciones. Cuando los semáforos se usan de esta manera, son similares a las variables flag, y su uso sigue los principios de sincronización con flags (3.16).

Podemos usar barreras de dos procesos como muestra esta solución para implementar una butterfly barrier para n procesos siguiendo la estructura ya vista. O podemos usar la misma idea para implementar una dissemination barrier. En ese caso, cada proceso señaliza el semáforo barrier de otro proceso, y luego espera a que su semáforo sea señalizado. Alternativamente, podemos usar semáforos como signal flags para implementar sincronización barrier n-proceso usando un proceso coordinador central o un combining tree. De hecho, dado que las operaciones **V** son recordadas, solo necesitamos un semáforo para el *Coordinator*. Esta es otra virtud de los semáforos respecto de los flags booleanos.

Productores y Consumidores: Split de Semáforos Binarios

Esta sección reexamina el problema de productores/consumidores introducido en el capítulo 2. Allí, asumíamos que había un productor y un consumidor; ahora consideramos la situación general en la cual hay múltiples productores y consumidores. La solución ilustra otro uso de los semáforos como flags de señalización. También introduce el concepto de un "split binary semaphore".

En este problema, los productores envían mensajes que son recibidos por los consumidores. Los procesos se comunican usando un buffer simple compartido, el cual es manipulado por dos operaciones: *deposit* y *fetch*. Los productores insertan mensajes en el buffer llamando a *deposit*; los consumidores reciben los mensajes llamando a *fetch*. Para asegurar que los mensajes no son sobrescritos antes de ser recibidos y que son recibidos solo una vez, la ejecución de *deposit* y *fetch* debe alternarse, con *deposit* ejecutándose primero.

Como con la sincronización barrier, la manera de especificar la propiedad de alternancia es usar contadores incrementales para indicar cuándo un proceso alcanza puntos de ejecución críticos. Aquí los puntos críticos son comenzar y completar la ejecución de *deposit* y *fetch*. Así, sean *inD* y *afterD* enteros que cuentan el número de veces que los productores han comenzado y terminado de ejecutar *deposit*. También, sean *inF* y *afterF* enteros que cuentan el número de veces que los consumidores han comenzado y terminado de ejecutar *fetch*. Entonces, el siguiente predicado especifica que *deposit* y *fetch* alternan:

PC: inD ≤ afterF + 1 ∧ inF ≤ afterD

En palabras, esto dice que *deposit* puede ser iniciado a lo sumo una vez más que las veces que se completó *fetch* y que *fetch* puede ser iniciado no más veces que las que se ha completado *deposit*.

Para este problema, las variables compartidas son los contadores y una variable *buf* que mantiene un mensaje de algún tipo T. Dado que nos interesa es cómo sincronizan los productores y consumidores, cada proceso simplemente ejecuta un loop; los productores repetidamente depositan mensajes, y los consumidores repetidamente los buscan. Los productores depositan mensajes en el buffer ejecutando:

```
deposit: ⟨ inD := inD + 1 ⟩
         buf := m
         ⟨ afterD := afterD + 1 ⟩
```

Los consumidores buscan los mensajes del buffer ejecutando:

```
fetch: ⟨ inF := inF + 1 ⟩
       m := buf
       ⟨ afterF := afterF + 1 ⟩
```

Para tener una implementación correcta de *deposit* y *fetch*, necesitamos guardar las asignaciones para asegurar la invarianza de la propiedad de sincronización PC. Nuevamente usamos *wp* para computar las guardas, vemos que necesitamos guardar los incrementos de *inD* e *inF* pero no los incrementos de *afterD* y *afterF* pues estos claramente preservan el invariante. (Además recordemos que nunca es necesario demorar cuando se deja una SC de código). Agregando las guardas que aseguran la invarianza de PC obtenemos la siguiente solución:

```
var buf : T       # para algún tipo T
var inD := 0, afterD := 0, inF := 0, afterF := 0
{ PC: inD ≤ afterF + 1 ∧ inF ≤ afterD }
Producer[i: 1..M]:: do true →
                        producir mensaje m
                        deposit: ⟨ await inD < afterF → inD := inD + 1 ⟩
                                 buf := m
                                 ⟨ afterD := afterD + 1 ⟩
                    od
Consumer[i: 1..N]:: do true →
```

```
                    fetch: ⟨ await inF < afterD → inF := inF + 1 ⟩
                              m := buf
                              ⟨ afterF := afterF + 1 ⟩
                    consumir mensaje m
                od
```

Para implementar las sentencias que acceden a los contadores usando semáforos, podemos usar la técnica de cambio de variables pues se cumplen las condiciones (4.2). En particular, sean *empty* y *full* semáforos cuyos valores son:

empty = afterF - inD + 1
full = afterD - inF

Con este cambio, los cuatro contadores se convierten en variables auxiliares, de modo que pueden ser borrados. Así, las primeras sentencias en *deposit* y *fetch* se convierten en operaciones **P**, y las últimas sentencias en operaciones **V**. Esto lleva a la solución final:

```
var buf : T       # para algún tipo T
var empty : sem := 1, full : sem := 0
{ PC' : 0 ≤ empty + full ≤ 1 }
Producer[i: 1..M]:: do true →
                    producir mensaje m
                    deposit: P(empty)
                              buf := m
                              V(full)
                od
Consumer[i: 1..N]:: do true →
                    fetch: P(full)
                              m := buf
                              V(empty)
                    consumir mensaje m
                od
```

En esta solución, *empty* y *full* son semáforos binarios. Más aún, juntos forman lo que se llama un “split binary semaphore”: a lo sumo uno de ellos es 1 a la vez, como especifica el predicado PC’.

(4.3) **Split Binary Semaphore**. Los semáforos binarios b_1, ..., b_n forman un split binary semaphore en un programa si la siguiente aserción es un invariante global en el programa: SPLIT: $0 \leq b_1$, + ... + $b_n \leq 1$.

El término *split binary semaphore* viene del hecho de que los b_i pueden verse como un único semáforo binario *b* que fue dividido en *n* semáforos binarios. Estos *n* semáforos se usan de manera tal que SPLIT es invariante.

Estos semáforos son importantes por la forma en que pueden usarse para implementar exclusión mutua. Dado un SBS, supongamos que uno de los semáforos que lo constituyen tiene un valor inicial de 1 (por lo tanto los otros son inicialmente 0). Además supongamos que, en los procesos que usan los semáforos, cualquier camino de ejecución alternativamente ejecuta una operación **P** sobre uno de los semáforos, luego una operación **V** sobre uno de los semáforos posiblemente distinto. Luego todas las sentencias entre cualquier **P** y el próximo **V** se ejecutan con exclusión mutua. Esto es porque, mientras un proceso está entre un **P** y un **V**, los semáforos son todos 0, y por lo tanto ningún otro proceso puede completar un **P** hasta que el primer proceso ejecute un **V**. La solución al problema de productores/consumidores ilustra esto: cada *Producer* alternativamente ejecuta **P**(empty) y luego **V**(full); cada *Consumer* ejecuta alternativamente **P**(full) y **V**(empty). Más adelante usaremos esta propiedad para construir un método general para implementar sentencias **await**.

Buffers Limitados: Contador de Recursos

La sección previa mostraba cómo sincronizar el acceso a un buffer de comunicación simple. Si los mensajes son producidos a una velocidad similar a la que son consumidos, un buffer simple provee una performance razonable pues un proceso generalmente no tendría que esperar demasiado para acceder al buffer. Sin embargo, comúnmente la ejecución del productor y el consumidor se da por ráfagas. Por ejemplo, un proceso que escribe en un archivo de salida podría producir varias líneas a la vez, y luego hacer más cómputo antes de producir otro conjunto de líneas. En tales casos, un buffer de mayor capacidad puede incrementar significativamente la performance reduciendo el número de veces que el proceso se bloquea.

Esta sección desarrolla una solución al problema del *buffer limitado*, es decir el problema de implementar un buffer de comunicación multislot. La solución se construye sobre la solución de la sección previa. También ilustra el uso de semáforos generales como contadores de recursos.

Asumimos por ahora que hay solo un productor y un consumidor. El productor deposita mensajes en un buffer compartido; el consumidor los busca. El buffer contiene una cola de mensajes que han sido depositados pero aún no buscados. Esta cola puede ser representada por una lista enlazada o por un arreglo. Usaremos un arreglo. En particular, sea el buffer representado por *buf*[1:n], donde *n* es mayor que 1. Sea *front* el índice del mensaje al principio de la cola, y sea *rear* el índice del primer lugar vacío. Inicialmente, *front* y *rear* son 0. Luego el productor deposita el mensaje *m* en el buffer ejecutando:

deposit: buf[rear] := m; rear := rear **mod** n + 1

Y el consumidor busca un mensaje en su variable local *m* ejecutando:

fetch: m := buf[front]; front := front **mod** n + 1

El operador **mod** se usa para asegurar que los valores de *front* y *rear* están siempre entre 1 y *n*. La cola de mensajes es almacenada en slots desde *buf*[front] hasta *buf*[rear] no inclusive, con *buf* tratado como un arreglo circular en el cual *buf*[1] sigue a *buf*[n].

Cuando hay un buffer simple la ejecución de *deposit* y *fetch* debe alternarse. Cuando hay múltiples buffers, *deposit* puede ejecutarse siempre que haya un slot vacío, y *fetch* puede ejecutarse siempre que haya un mensaje almacenado. Podemos especificar este requerimiento de sincronización por una generalización directa del predicado PC de productores/consumidores. Como antes, *inD* y *afterD* indican el número de veces que el productor ha comenzado y terminado de ejecutar *deposit*, e *inF* y *afterF* indican el número de veces que el consumidor ha comenzado y terminado de ejecutar *fetch*. Luego el acceso a *buf* necesita ser sincronizado de modo que el siguiente predicado sea un invariante global:

BUFFER: inD $\leq$ afterF + n $\wedge$ inF $\leq$ afterD

La única diferencia entre este predicado y PC de la sección anterior es que *deposit* puede comenzar hasta *n* veces más que las que finalizó *fetch*, en lugar de solo una vez más. Insertando asignaciones a estas variables al comienzo y el final de *deposit* y *fetch* y guardando aquellas asignaciones para asegurar que BUFFER es invariante, tenemos la siguiente solución:

```
var buf[1:n] : T        # para algún tipo T
var front := 0, rear := 0
var inD := 0, afterD := 0, inF := 0, afterF := 0
{ BUFFER: inD ≤ afterF + n ∧ inF ≤ afterD }
Producer:: do true →
                producir mensaje m
                deposit: ⟨ await inD < afterF + n → inD := inD + 1 ⟩
                        buf[rear] := m; rear := rear mod n + 1
                        ⟨ afterD := afterD + 1 ⟩
           od
Consumer:: do true →
                fetch: ⟨ await inF < afterD → inF := inF + 1 ⟩
```

```
                    m := buf[front]; front := front mod n + 1
                    ⟨ afterF := afterF + 1 ⟩
                consumir mensaje m
            od
```

En esta solución, *Producer* y *Consumer* pueden de hecho acceder *buf* al mismo tiempo. Esto es perfectamente aceptable dado que, si *deposit* y *fetch* ejecutan concurrentemente, entonces *front* ≠ *rear*, de modo que los procesos estarán accediendo distintos elementos del arreglo *buf*.

Nuevamente podemos usar la técnica de cambio de variables para implementar las acciones atómicas de esta solución usando semáforos. Nuevamente sean *empty* y *full* semáforos que indican el número de slots vacíos y llenos, respectivamente. Están relacionados con los contadores incrementales por:

```
empty = afterF - inD + n
full = afterD - inF
```

Con este cambio de variables, las acciones atómicas nuevamente se convierten en operaciones sobre semáforos, y tenemos la siguiente solución final, en la cual los semáforos son usados de la misma manera que antes (la única diferencia es que *empty* se inicializa en *n*):

```
var buf[1:n] : T       # para algún tipo T
var front := 0, rear := 0
var empty : sem := n, full : sem := 0          # n - 2 ≤ empty + full ≤ n
{ BUFFER: inD ≤ afterF + n ∧ inF ≤ afterD }
Producer:: do true →
                producir mensaje m
                deposit: P(empty)
                        buf[rear] := m; rear := rear mod n + 1
                        V(full)
            od
Consumer:: do true →
                fetch: P(full)
                        m := buf[front]; front := front mod n + 1
                        V(empty)
                consumir mensaje m
            od
```

En esta solución, los semáforos sirven como *contadores de recursos*: cada uno cuenta el número de unidades de un recurso. En este caso, *empty* cuenta el número de slots vacíos, y *full* cuenta el número de slots llenos. Cuando ningún proceso está ejecutando *deposit* y *fetch*, la suma de los valores de los dos semáforos es *n*, el número total de slots del buffer. Los semáforos contadores de recursos son útiles cuando los procesos compiten por el acceso a recursos de múltiple unidad tales como los slots del buffer o bloques de memoria.

En esta solución, asumimos que hay solo un productor y un consumidor pues asegura que *deposit* y *fetch* se ejecutan como acciones atómicas. Supongamos que hay dos (o más) productores. Luego cada uno podría estar ejecutando *deposit* al mismo tiempo si hay al menos dos slots vacíos. En ese caso, ambos procesos podrían tratar de depositar su mensaje en el mismo slot! (Esto ocurre si ambos asignan a buf[rear] antes de que cualquiera incremente rear). Similarmente, si hay dos (o más) consumidores, ambos podrían ejecutar *fetch* al mismo tiempo y obtener el mismo mensaje. Es decir: *deposit* y *fetch* se convierten en secciones críticas. Cada uno debe ser ejecutado con exclusión mutua (pero pueden ejecutar concurrentemente uno con otro pues *empty* y *full* se usan de tal manera que los productores y consumidores acceden distintos slots del buffer. Podemos implementar la exclusión requerida usando la solución al problema de la SC visto en este capítulo, con semáforos separados usados para proteger cada SC. La solución completa es la siguiente:

```
var buf[1:n] : T       # para algún tipo T
var front := 0, rear := 0
var empty : sem := n, full : sem := 0          # n - 2 ≤ empty + full ≤ n
var mutexD : sem := 1, mutexF : sem := 1
```

```
Producer[1:M]:: do true →
                    producir mensaje m
                    deposit: P(empty)
                             P(mutexD)
                             buf[rear] := m; rear := rear mod n + 1
                             V(mutexD)
                             V(full)
                od
Consumer[1:N]:: do true →
                     fetch: P(full)
                            P(mutexF)
                            m := buf[front]; front := front mod n + 1
                            V(mutexF)
                            V(empty)
                     consumir mensaje m
                od
```

Aquí hemos resuelto los dos problemas separadamente (primero la sincronización entre el productor y el consumidor, luego la sincronización entre productores y entre consumidores. Esto hizo simple combinar las soluciones a los dos subproblemas para obtener una solución al problema completo. Usaremos la misma idea para resolver el problema de lectores/escritores. Cuando hay múltiples clases de sincronización, generalmente es útil examinarlos separadamente y luego combinar las soluciones.

EXCLUSION MUTUA SELECTIVA

La sección anterior mostró como usar semáforos para implementar secciones críticas. Esta sección se basa en esa técnica para implementar formas más complejas de exclusión mutua. Consideramos dos problemas de sincronización clásicos: filósofos y lectores/escritores. La solución al problema de los filósofos ilustra cómo implementar exclusión mutua entre procesos que compiten por el acceso a conjuntos superpuestos de variables compartidas. La solución al problema de lectores/escritores ilustra como implementar una combinación de acceso concurrente y exclusivo a variables compartidas.

El problema de los filósofos

Aunque es un problema más teórico que práctico, es similar a problemas realistas en los cuales un proceso requiere acceso simultáneo a más de un recurso. En consecuencia, el problema es usado con frecuencia para ilustrar y comparar distintos mecanismos de sincronización.

(4.4) **El problema de los filósofos**. Cinco filósofos se sientan alrededor de una mesa circular. Cada uno pasa su vida pensando y comiendo. En el centro de la mesa hay un gran plato de spaghetti. Cada filósofo debe usar dos tenedores para comer. Desafortunadamente, los filósofos tiene solo 5 tenedores. Un tenedor se ubica entre cada par de filósofos, y acuerdan que cada uno usará solo los tenedores a su inmediata izquierda y derecha. El problema es escribir un programa para simular el comportamiento de los filósofos. Este programa debe evitar la desafortunada (y eventualmente fatal) situación en la cual todos los filósofos están hambrientos pero ninguno es capaz de adquirir ambos tenedores.

Claramente, dos filósofos vecinos no pueden comer al mismo tiempo. Además, con solo 5 tenedores, a lo sumo dos filósofos a la vez pueden estar comiendo.

Simularemos las acciones de los filósofos como sigue:

```
Philosopher[i:1..5] :: do true →
                           piensa
                           adquiere tenedores
                           come
```

libera tenedores
od

Asumimos que la longitud de los períodos en que los filósofos están pensando y comiendo varían; esto podría simularse usando demoras random.

Resolver el problema requiere programar las acciones de adquirir y liberar los tenedores. Como es usual, el primer paso es especificar la sincronización requerida precisamente. Dado que los tenedores son los recursos compartidos, nos concentraremos en las acciones de adquirirlos y liberarlos. (Alternativamente, podríamos especificar la sincronización en términos de si los filósofos están o no comiendo).

Para especificar los estados de los filósofos y los tenedores, podríamos registrar para cada filósofo y tenedor si un filósofo tiene un tenedor o no. Esto es similar a especificar si un proceso está o no en su SC. O podemos usar nuevamente contadores incrementales; esto resulta en una especificación más simple pues usa menos variables. Sea que *up*[i] cuenta el número de veces que el tenedor *i* fue levantado, y *down*[i] cuenta el número de veces que fue bajado. Claramente un tenedor no puede bajarse más veces de las que se levantó. Además, un tenedor puede se levantado por a lo sumo un filósofo a la vez. Estas restricciones son especificadas por el predicado:

FORKS: ($\forall$ i: $1 \leq i \leq 5$: down[i] $\leq$ up[i] $\leq$ down[i] + 1)

Este debe ser un invariante global.

Cada filósofo primero levanta dos tenedores, luego come, luego baja los tenedores. Asumimos que el filósofo *i* levanta los tenedores *i* y (*i* **mod** 5) + 1. Con esta representación, levantar un tenedor involucra incrementar el elemento apropiado de *up*, y bajar el tenedor involucra incrementar el elemento apropiado de *down*. Guardando los incrementos a los contadores para asegurar la invarianza de FORKS obtenemos la siguiente solución coarse-grained (usamos $i \oplus 1$ en lugar de (*i* **mod** 5) + 1:

```
var up[1:5] : int := ( [5] 0 ), down[1:5] int := ( [5] 0 )
{ FORKS: ( ∀ i: 1 ≤ i ≤ 5: down[i] ≤ up[i] ≤ down[i] + 1 ) }
Philosopher[i:1..5]:: do true →
                          ⟨ await up[i] = down[i] → up[i] := up[i] + 1 ⟩
                          ⟨ await up[i ⊕ 1] = down[i ⊕ 1] → up[i ⊕ 1] := up[i ⊕ 1] +
1⟩
                          come
                          ⟨ down[i] := down[i] + 1 ⟩
                          ⟨ down[i ⊕ 1] := down[i ⊕ 1] + 1⟩
                          piensa
                      od
```

Nuevamente las sentencias atómicas cumplen los requerimientos para realizar cambio de variables, de modo que *up* y *down* pueden ser reemplazadas por un nuevo arreglo *fork* que satisface:

fork[i] = 1 - (up[i] - down[i])

Como antes, *up* y *down* se convierten en variables auxiliares, y *fork* se convierte en un arreglo de semáforos. *Philosopher*[i] levanta ambos tenedores ejecutando:

P(fork[i]); **P**(fork[i $\oplus$ 1])

Similarmente, baja los tenedores ejecutando:

V(fork[i]); **V**(fork[i $\oplus$ 1])

Aunque esta aproximación asegura que los filósofos vecinos no coman al mismo tiempo, la solución puede resultar en deadlock. En particular, si todos los filósofos levantaron un tenedor, luego ninguno puede levantar su segundo. Una condición necesaria para deadlock es que hay waiting circular, es decir, un proceso está esperando un recurso tomado por un segundo, el cual espera un recurso tomado por un tercero, y así siguiendo hasta un proceso que espera un recurso tomado por el primero. Así, para evitar deadlock es suficiente asegurar que no puede ocurrir waiting circular. Para este problema, una aproximación es tener uno de los procesos, digamos *Philosopher*[5], levantando sus tenedores en orden inverso. La solución es la siguiente (probar formalmente que la solución está libre de deadlock requiere agregar variables auxiliares para indicar cuál filósofo toma cada tenedor, y luego usar el método de Exclusión de Configuraciones):

```
var forks[1:5] : sem := ( [5] 1 )
Philosopher[i:1..4]:: do true →
                          P(fork[i]); P(fork[i ⊕ 1])
                          come
                          V(fork[i]); V(fork[i ⊕ 1])
                          piensa
                      od
Philosopher[5]:: do true →
                          P(fork[1]); P(fork[5])
                          come
                          V(fork[1]); V(fork[5])
                          piensa
                 od
```

Lectores y Escritores

El problema de lectores y escritores es otro problema de sincronización clásico. Como los filósofos, con frecuencia se usa para comparar y contrastar mecanismos de sincronización. Además es un problema eminentemente práctico.

(4.5) **Problema de Lectores/Escritores**. Dos clases de procesos (lectores y escritores) comparten una base de datos. Los lectores ejecutan transacciones que examinan registros de una BD; las transacciones de los escritores examinan y actualizan la BD. La BD se asume que inicialmente está en un estado consistente. Cada transacción, si es ejecutada aisladamente, transforma la BD de un estado consistente a otro. Para evitar interferencia entre transacciones, un proceso escritor debe tener acceso exclusivo a la BD. Asumiendo que ningún escritor está accediendo la BD, cualquier número de lectores puede ejecutar transacciones concurrentemente.

Es otro ejemplo de problema de exclusión mutua selectiva. En el problema de los filósofos, pares de procesos competían para acceder a los tenedores. Aquí, clases de procesos compiten por el acceso a la BD. Los procesos escritores individuales compiten por el acceso con cada uno de los otros, y los lectores como una clase compiten con los escritores.

Aquí derivamos una solución al problema que ilustra cómo implementar esta clase de exclusión. Como es usual, el punto de partida es especificar la sincronización requerida. Dado que los escritores necesitan excluir a cada uno de los otros, podemos especificar su interacción como en el problema de la SC. En particular, sea *writing*[j] = 1 cuando un escritor está accediendo la BD, y *writing*[j] = 0 en otro caso. Luego, la suma de los *writing*[j] debe ser a lo sumo 1. Dado que los lectores como clase necesitan excluir escritores individuales, es suficiente emplear una variable, *reading*, que es 1 cuando cualquier lector está leyendo la BD y es 0 en otro caso. Inicialmente todas las variables son 0. El siguiente predicado especifica la sincronización requerida (y como es usual debe ser un invariante global):

RW: reading + writing[1] + ... + writing[n] ≤ 1

La especificación anterior dice que *reading* es 1 cuando cualquier lector está leyendo la BD y 0 en otro caso. Esto es diferente que en problemas de exclusión mutua previos. Antes, había una restricción sobre cada proceso individual. Aquí, solo el primer lector debería incrementar *reading* antes de acceder la BD, y solo el último lector debería decrementarla cuando todos los lectores terminaron. Para implementar esto, sea *nr* el número de lectores activos (es decir, aquellos que están tratando de acceder la BD). Luego, *reading* debería ser incrementada cuando *nr* es 1 y decrementada cuando *nr* es nuevamente 0. Ignorando la representación real de la BD (la cual en general está almacenada en archivos) e ignorando otras acciones de los procesos, tenemos el siguiente outline de solución:

```
var reading := 0, nr := 0, writing[1:n] := ( [n] 0 )
{ RW: reading + writing[1] + .... + writing[n] ≤ 1 }
Reader[i:1..m] :: do true →
                        ⟨ nr := nr + 1
                          if nr = 1 → reading := reading + 1 fi ⟩
                        lee la BD
                        ⟨ nr := nr - 1
                          if nr = 0 → reading := reading - 1 fi ⟩
                  od
Writer[j:1..n] :: do true →
                        ⟨ writing[j] := writing[j] + 1 ⟩
                        escribe la BD
                        ⟨ writing[j] := writing[j] - 1 ⟩
                  od
```

En los lectores, alterar y examinar *nr* y *reading* se hace en acciones atómicas compuestas para preservar la relación especificada entre estas acciones). En particular, fuera de las acciones atómicas, el siguiente predicado es también invariante:

$$(nr > 0 \wedge reading = 1) \vee (nr = 0 \wedge reading = 0)$$

Para refinar esto en una solución que use semáforos, primero necesitamos guardar las asignaciones a *reading* y *writing* para asegurar que RW es invariante. Luego necesitamos implementar las acciones atómicas resultantes. Para asegurar RW, cuando *reading* es incrementada, la suma de *writing*[j] debe ser 0. Cuando *writing*[j] es incrementada, la suma de *reading* y los otros *writing*[k] debe ser 0. Como es usual, decrementar *reading* y *writing*[j] no necesita ser guardada. Agregando guardas apropiadas a la solución anterior obtenemos la siguiente solución coarse-grained:

```
var reading := 0, nr := 0, writing[1:n] := ( [n] 0 )
{ RW: reading + writing[1] + .... + writing[n] ≤ 1 }
# debajo SUM es reading + writing[1] + .... + writing[n]
Reader[i:1..m] :: do true →
                        ⟨ nr := nr + 1
                          if nr = 1 →
                                await SUM ≤ 0 → reading := reading + 1 fi ⟩
                        lee la BD
                        ⟨ nr := nr - 1
                          if nr = 0 → reading := reading - 1 fi ⟩
                  od
Writer[j:1..n] :: do true →
                        ⟨ await SUM ≤ 0 → writing[j] := writing[j] + 1 ⟩
                        escribe la BD
                        ⟨ writing[j] := writing[j] - 1 ⟩
                  od
```

La sentencia **await** en los lectores está en el medio de una acción atómica. Es la primera vez que ocurrió esto. En consecuencia, necesitamos asegurar que la acción entera es indivisible, aún si **await** causa demora. Debajo se muestra cómo hacer esto.

Recordemos que, si dos acciones atómicas referencian conjuntos disjuntos de variables, entonces pueden ejecutar concurrentemente pues aparecerán como indivisibles con respecto a la otra. En la solución, *nr* es referenciada solo por los procesos lectores. Las únicas acciones dentro de los lectores que referencian variables accedidas por los escritores son la sentencia **await** y el decremento de *reading*. Por lo tanto, las acciones compuestas en los lectores pueden ser subdivididas en las partes que son críticas con respecto a otros lectores y las partes que también son críticas con respecto a los escritores. Ya mostramos cómo usar semáforos para implementar SC. Sea *mutexR* un semáforo que se usa para implementar SC entre lectores. Usando este semáforo, el protocolo de entrada de los lectores se convierte en:

```
P(mutexR)
  nr := nr + 1
  if nr = 1 → ⟨ await SUM ≤ 0 → reading := reading + 1⟩ fi
V(mutexR)
```

El correspondiente protocolo de salida para el lector es:

```
P(mutexR)
  nr := nr - 1
  if nr = 0 → ⟨ reading := reading - 1 ⟩ fi
V(mutexR)
```

En esencia, reemplazamos los corchetes por operaciones sobre semáforos. Luego de hacer esto, necesitamos encerrar las acciones que referencian variables compartidas con los escritores (*reading* y los *writing*[j]) dentro de corchetes para indicar que también deben ser atómicas.

Para obtener una solución completa, ahora necesitamos usar semáforos para implementar acciones de lectores y escritores. Dado que estas acciones implementan exclusión lector/escritor (como lo especifica que RW sea invariante) nuevamente podemos usar la técnica de cambio de variables. En particular, sea *rw* un semáforo cuyo valor es:

```
rw = 1 - (reading + writing[1] + .... + writing[n])
```

Haciendo este cambio, las acciones atómicas restantes se convierten en operaciones sobre semáforos, obteniendo la siguiente solución final:

```
var nr := 0, mutexR : sem := 1, rw : sem := 1
Reader[i:1..m] :: do true →
                      P(mutexR)
                        nr := nr + 1
                        if nr = 1 → P(rw) fi
                      V(mutexR)
                      lee la BD
                      P(mutexR)
                        nr := nr - 1
                        if nr = 0 → V(rw) fi
                      V(mutexR)
                  od
Writer[j:1..n] :: do true →
                      P(rw)
                      escribe en la BD
                      V(rw)
                  od
```

En esta solución, *mutexR* protege la SC entre lectores, y *rw* protege la SC entre lectores y escritores. La variable *nr* se usa de modo tal que solo el primer lector que arriba ejecuta **P**(rw), y el último lector que se va ejecuta **V**(rw).

Este algoritmo es llamado de *preferencia de los lectores*. Este término denota el hecho de que, si algún lector está accediendo la BD y llegan un lector y un escritor a sus entry protocols, entonces el nuevo lector tiene preferencia sobre el escritor. Realmente, la solución da a los lectores preferencia “débil”. Una solución de preferencia “fuerte” de los lectores es una en la cual un lector *siempre* tiene preferencia sobre un escritor cuando ambas clases de procesos están en sus entry protocols (en particular, si un lector y un escritor están demorados ejecutando **P**(rw)).

Dado que el algoritmo da a los lectores preferencia sobre los escritores, la solución no es fair. Esto es porque una serie continua de lectores puede evitar permanentemente que los escritores accedan la BD. La próxima sección desarrolla una solución diferente que es fair.

SINCRONIZACION POR CONDICION GENERAL

La sección anterior aproximaba el problema de lectores/escritores como un problema de exclusión mutua. El foco estaba en asegurar que los escritores excluían a cada uno de los otros, y que los lectores como clase excluían a los escritores. La solución resultante consistía de overlappear soluciones a problemas de SC: uno entre lectores y otro entre escritores.

Esta sección desarrolla una solución distinta al problema partiendo de una especificación diferente (y más simple) de la sincronización requerida. La solución introduce una técnica de programación general llamada *passing the baton*. Esta técnica emplea split binary semaphores para proveer exclusión y controlar cuál proceso demorado es el próximo en seguir. Puede usarse para implementar sentencias **await** arbitrarias y así implementar sincronización por condición arbitraria. La técnica también puede usarse para controlar precisamente el orden en el cual los procesos demorados son despertados.

Lectores y escritores revisitado

Como definimos en (4.5) los lectores examinan una BD compartida, y los escritores la examinan y alteran. Para preservar la consistencia de la BD, un escritor requiere acceso exclusivo, pero cualquier número de lectores pueden ejecutar concurrentemente. Una manera simple de especificar esta sincronización es contar el número de cada clase de proceso que trata de acceder a la BD, luego restringir los valores de los contadores. En particular, sean *nr* y *nw* enteros no negativos que registran respectivamente el número de lectores y escritores que están accediendo la BD. El estado malo a evitar es uno en el cual tanto *nr* como *nw* son positivos o *nw* es mayor que uno. El conjunto inverso de estados buenos es caracterizado por el predicado:

$$RW: (nr = 0 \vee nw = 0) \wedge nw \leq 1$$

El primer término dice que los lectores y escritores no pueden acceder la BD al mismo tiempo; el segundo dice que hay a lo sumo un escritor activo. Con esta especificación del problema, un outline de los procesos lectores es:

```
Reader[i:1..m] :: do true → ⟨ nr := nr + 1 ⟩
                              lee la BD
                              ⟨ nr := nr - 1 ⟩
                  od
```

Para los escritores:

```
Writer[j:1..n] :: do true → ⟨ nw := nw + 1 ⟩
                             escribe la BD
                             ⟨ nw := nw - 1 ⟩
                  od
```

Para refinar esto en una solución coarse-grained, necesitamos guardar las asignaciones a las variables compartidas para asegurar que RW es invariante. A partir de

wp(nr := nr + 1, RW) = (nr = -1 $\vee$ nw = 0)

y el hecho de que *nr* y *nw* son no negativos, necesitamos guardar nr := nr + 1 por nw = 0. Similarmente, necesitamos guardar nw := nw + 1 por nr = 0 $\wedge$ nw = 0. Sin embargo, no necesitamos guardar ningún decremento. Informalmente, nunca es necesario demorar un proceso que está devolviendo el uso de un recurso. Más formalmente,

wp(nr := nr - 1, RW) = ((nr = 1 $\vee$ nw = 0) $\wedge$ nw $\leq$ 1)

Esto es true pues (nr > 0 $\wedge$ RW) es true antes de que *nr* sea decrementada. El razonamiento para *nw* es análogo. Insertando las guardas obtenemos la siguiente solución coarse-grained:

```
var nr := 0, nw := 0
{ RW: ( nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1 }
Reader[i:1..m] :: do true → ⟨ await  nw = 0 → nr := nr + 1 ⟩
                              lee la BD
                              ⟨ nr := nr - 1 ⟩
                  od
Writer[j:1..n] :: do true → ⟨ await  nr = 0 and nw = 0 → nw := nw + 1 ⟩
                            escribe la BD
                            ⟨ nw := nw - 1 ⟩
                  od
```

La técnica de Passing the Baton

En la solución anterior, las dos guardas en las sentencias **await** se superponen. Por lo tanto, no podemos usar la técnica de cambio de variables para implementar las sentencias atómicas. Esto es porque ningún semáforo podría discriminar entre las guardas. Necesitamos otra técnica, que llamaremos *passing the baton*. Esta técnica es lo suficientemente poderosa para implementar cualquier sentencia **await**.

Después del tercer paso de nuestro método de derivación, la solución contendrá sentencias atómicas con una de las siguientes formas:

$F_1 : \langle S_i \rangle$ o $F_2 : \langle$ **await** $B_j \rightarrow S_j \rangle$

Podemos usar split binary semaphores como sigue para implementar tanto la exclusión mutua como sincronización por condición en estas sentencias. Primero, sea *e* un semáforo binario cuyo valor inicial es 1. Se usa para controlar la entrada a sentencias atómicas. Segundo, asociar un semáforo b_j y un contador d_j cada uno con guarda semánticamente diferente B_j; estos son todos inicialmente 0. El semáforo b_j se usa para demorar procesos esperando que B_j se convierta en true; d_j es un contador del número de procesos demorados (o cerca de demorarse) sobre b_j.

El conjunto entero de semáforos (*e* y los b_j) se usan como sigue para formar un split binary semaphore. Las sentencias de la forma F_1 se reemplazan por el fragmento de programa:

(4.6)
```
F1: P(e)      { I }
    Si        { I }
    SIGNAL
```

Las sentencias de la forma F_2 se reemplazan por el fragmento de programa:

(4.7)
```
F2: P(e)      { I }
    if not Bj → dj := dj + 1; V(e); P(bj) fi    { I ∧ Bj }
    Si        { I }
    SIGNAL
```

Hemos comentado los fragmentos de programa anteriores con aserciones que son true en puntos críticos; I es el invariante de sincronización. En ambos fragmentos, SIGNAL es la siguiente sentencia:

(4.8) SIGNAL: **if** B_1 **and** $d_1 > 0 \rightarrow$ { $I \wedge B_1$ } $d_1 := d_1 - 1$; **V**(b_1)
□ ...
□ B_n **and** $d_n > 0 \rightarrow$ { $I \wedge B_n$ } $d_n := d_n - 1$; **V**(b_n)
□ **else** $\rightarrow$ { I } **V**(e)
fi

Las primeras *n* guardas en SIGNAL chequean si hay algún proceso esperando por una condición que ahora es true. La última guarda, **else**, es una abreviación para la negación de la disyunción de las otras guardas (es decir, **else** es true si ninguna de las otras guardas es true). Si la guarda **else** es seleccionada, el semáforo de entrada *e* es señalizado. Nuevamente, SIGNAL está comentado con aserciones que son true en puntos críticos.

Con estos reemplazos, los semáforos forman un split binary semaphore pues a lo sumo un semáforo a la vez es 1, y cada camino de ejecución comienza con un **P** y termina con un único **V**. El invariante de sincronización I es true antes de cada operación **V**, de modo que es true cada vez que uno de los semáforos es 1. Más aún, B_j se garantiza que es true siempre que S_j es ejecutada. Esto es porque, o el proceso chequeó B_j y la encontró true, o el proceso se demoró sobre b_j, la cual es señalizada solo cuando B_j es true. En el último caso, el predicado B_j es transferido efectivamente al proceso demorado. Finalmente, la transformación no introduce deadlock pues b_j es señalizada solo si algún proceso está esperando o está a punto de esperar sobre b_j. (Un proceso podría haber incrementado el contador y ejecutado **V**(e) pero podría aún no haber ejecutado la operación **P** sobre el semáforo condición).

Esta técnica se llama *passing the baton* por la manera en que los semáforos son señalizados. Cuando un proceso está ejecutando dentro de una región crítica, podemos pensar que mantiene el baton que significa permiso para ejecutar. Cuando ese proceso alcanza un fragmento SIGNAL, pasa el baton a otro proceso. Si algún proceso está esperando una condición que ahora es verdadera, el baton es paso a tal proceso, el cual a su turno ejecuta la región crítica y pasa el baton a otro proceso. Cuando ningún proceso está esperando una condición que es true, el baton es pasado al próximo proceso que trata de entrar a su región crítica por primera vez (es decir, un proceso que ejecuta **P**(e)).

Solución a lectores y escritores

Podemos usar la técnica de passing the baton para implementar la solución coarse-grained al problema de lectores/escritores. En esa solución, hay dos guardas distintas, de modo que necesitamos dos semáforos de condición y contadores asociados. Sea que el semáforo *r* representa la condición de demora del lector *nw* = 0, y *w* representa la condición de demora del escritor *nr* = 0 $\wedge$ *nw* = 0. Sean *dr* y *dw* los contadores asociados. Finalmente, sea *e* el semáforo de entrada. Realizando los reemplazos de baton-passing dados en (4.6) y (4.7) obtenemos la siguiente solución:

```
var nr := 0, nw := 0    { RW: ( nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1 }
var e : sem := 1, r : sem := 0, w : sem := 0     {SPLIT: 0 ≤ (e + r + w) ≤ 1 }
var dr := 0, dw := 0    {COUNTERS: dr ≥ 0 ∧ dw ≥ 0 }
Reader[i:1..m] :: do true →
                        P(e)
                        if nw > 0 → dr := dr + 1; V(e); P(r) fi
                        nr := nr + 1
                        SIGNAL1
                        lee la BD
                        P(e)
                        nr := nr - 1
                        SIGNAL2
                  od
Writer[j:1..n] :: do true →
                        P(e)
```

```
                    if nr > 0 or nw > 0 → dw := dw + 1; V(e); P(w) fi
                    nw := nw + 1
                    SIGNAL3
                    escribe la BD
                    P(e)
                    nw := nw - 1
                    SIGNAL4
                od
```

Aquí $SIGNAL_i$ es una abreviación de:

```
if nw = 0 and dr > 0 → dr := dr - 1; V(r)
□ nr = 0 and nw = 0 and dw > 0 → dw := dw - 1; V(w)
□ (nw > 0 or dr = 0 and (nr > 0 or nw > 0 od dw = 0) → V(e)
fi
```

Aquí, $SIGNAL_i$ asegura que nw es 0 cuando el semáforo *r* es señalizado y asegura que tanto nr como nw son 0 cuando el semáforo *w* es señalizado. El semáforo *e* es señalizado solo cuando no hay lectores o escritores demorados que podrían seguir.

En la solución anterior (y en general) las precondiciones de los fragmentos SIGNAL permiten que varias de las guardas sean simplificadas o eliminadas. En los procesos lectores, tanto nr > 0 como nw = 0 son true antes de $SIGNAL_1$. Por lo tanto ese fragmento signal se simplifica a:

```
if dr > 0 → dr := dr -1; V(r)  □  dr = 0 →  V(e) fi
```

Antes de $SIGNAL_2$ en los lectores, tanto nw como dr son 0. En los escritores, nr = 0 y nw > 0 antes de $SIGNAL_3$, y nr = 0 y nw = 0 antes de $SIGNAL_4$. Usando estos hechos para simplificar los protocolos de señalización obtenemos la siguiente solución final:

```
var nr := 0, nw := 0    { RW: ( nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1 }
var e : sem := 1, r : sem := 0, w : sem := 0     {SPLIT: 0 ≤ (e + r + w) ≤ 1 }
var dr := 0, dw := 0    {COUNTERS: dr ≥ 0 ∧ dw ≥ 0 }
Reader[i:1..m] :: do true →
                    P(e)
                    if nw > 0 → dr := dr + 1; V(e); P(r) fi
                    nr := nr + 1
                    if dr > 0 → dr := dr - 1; V(r)  □ nr = 0 → V(e) fi
                    lee la BD
                    P(e)
                    nr := nr - 1
                    if nr = 0 and dw > 0 → dw := dw - 1; V(w)
                    □ nr > 0 or dw = 0 → V(e)
                    fi
                od
Writer[j:1..n] :: do true →
                    P(e)
                    if nr > 0 or nw > 0 → dw := dw + 1; V(e); P(w) fi
                    nw := nw + 1
                    V(e)
                    escribe la BD
                    P(e)
                    nw := nw - 1
                    if dr > 0 → dr := dr - 1; V(r)
                    □ dw > 0 → dw := dw - 1; V(w)
                    □ dr = 0 and dw = 0 → V(e)
                    fi
                od
```

En esta solución, la última sentencia **if** en los escritores es no determinística. Si hay lectores y escritores demorados, cualquiera podría ser señalizado cuando un escritor termina su exit protocol. Además, cuando finaliza un escritor, si hay más de un lector demorado y uno es despertado, los otros son despertados en forma de "cascada". El primer lector incrementa *nr*, luego despierta al segundo, el cual incrementa *nr* y despierta al tercero, etc. El baton se va pasando de un lector demorado a otro hasta que todos son despertados.

Políticas de Scheduling Alternativas

Como en la primera solución que dimos al problema de los filósofos, esta solución da a los lectores preferencia sobre los escritores. Sin embargo, a diferencia de la primera solución rápidamente podemos modificar esta para realizar el scheduling de los procesos de otras maneras. Por ejemplo, para dar preferencia a los escritores, es necesario asegurar que:

- nuevos lectores son demorados si un escritor está esperando, y
- un lector demorado es despertado solo si ningún escritor está esperando

Podemos cumplir el primer requerimiento fortaleciendo la condición de demora en la primera sentencia **if** de los lectores:

if nw > 0 **or** dw > 0 → dr := dr + 1; **V**(e); **P**(r) **fi**

Para cumplir el segundo requerimiento, podemos fortalecer la primera guarda en la última sentencia **if** de los escritores:

if dr > 0 **and** dw = 0 → dr := dr - 1; **V**(r)
□ dw > 0 → dw := dw - 1; **V**(w)
□ dr = 0 **and** dw = 0 → **V**(e)
fi

Esto elimina el no determinismo que estaba presente en esa sentencia, lo cual siempre es seguro. Ninguno de estos cambios altera la estructura de la solución. Esta es una virtud de la técnica de passing the baton: las guardas pueden ser manipuladas para alterar el orden en el cual los procesos son despertados sin afectar la corrección básica de la solución.

También podemos alterar la solución para asegurar acceso fair a la BD, asumiendo que las operaciones sobre semáforos son en si mismas fair. Por ejemplo, podríamos forzar a los lectores y escritores a alternar los turnos cuando ambos están esperando. En particular, cuando un escritor termina, todos los lectores esperando toman un turno; y cuando los lectores terminan, un escritor esperando toma el turno. Podemos implementar esta alternancia agregando una variable booleana *writer_last* que se setea en true cuando un escritor comienza a escribir y es limpiada cuando un lector comienza a leer. Entonces cambiamos la última sentencia **if** en los escritores:

if dr > 0 **and** (dw = 0 or writer_last) → dr := dr - 1; **V**(r)
□ dw > 0 **and** (dr = 0 **or not** writer_last) → dw := dw - 1; **V**(w)
□ dr = 0 **and** dw = 0 → **V**(e)
fi

Nuevamente la estructura de la solución no cambia.

Esta técnica de passing the baton puede ser usada también para proveer control finer-grained sobre el orden en el cual los procesos usan recursos. La próxima sección ilustra esto. Lo único que no podemos controlar es el orden en el cual los procesos demorados en el semáforo *e* son despertados. Esto depende de la implementación subyacente de los semáforos.

ALOCACION DE RECURSOS

La alocación de recursos es el problema de decidir cuándo se le puede dar a un proceso acceso a un recurso. En programas concurrentes, un recurso es cualquier cosa por la que un proceso podría ser demorado esperando adquirirla. Esto incluye entrada a una SC, acceso a una BD, un slot en un buffer limitado, una región de memoria, el uso de una impresora, etc. Ya hemos examinado varios problemas de alocación de recursos específicos. En la mayoría, se empleó la política de alocación posible más simple: si algún proceso está esperando y el recurso está disponible, se lo aloca. Por ejemplo, la solución al problema de la SC aseguraba que se le daba permiso para entrar a *algún* proceso que estaba esperando; no intentaba controlar a cuál proceso se le daba permiso si había una elección. De manera similar, la solución al problema del buffer limitado no intentaba controlar cuál productor o cuál consumidor eran el próximo en acceder al buffer. La política de alocación más compleja que consideramos fue en el problema de lectores/escritores. Sin embargo, nuestra atención estuvo en darle preferencia a clases de procesos, no a procesos individuales.

Esta sección muestra cómo implementar políticas de alocación de recursos generales y en particular muestra cómo controlar explícitamente cuál proceso toma un recurso cuando hay más de uno esperando. Primero describimos el patrón de solución general. Luego implementamos una política de alocación específica (shortest job next). La solución emplea la técnica de passing the baton. También introduce el concepto de semáforos privados, lo cual provee la base para resolver otros problemas de alocación de recursos.

Definición del problema y Patrón de solución general

En cualquier problema de alocación de recursos, los procesos compiten por el uso de unidades de un recurso compartido. Un proceso pide una o más unidades ejecutando la operación *request*, la cual con frecuencia es implementada por un procedure. Los parámetros a *request* indican cuantas unidades se requieren, identifican alguna característica especial tal como el tamaño de un bloque de memoria, y dan la identidad del proceso que pide. Cada unidad del recurso compartido está libre o en uso. Un pedido puede ser satisfecho cuando todas las unidades del recurso compartido están libres. Por lo tanto *request* se demora hasta que esta condición es true, luego retorna el número requerido de unidades. Después de usar los recursos alocados, un proceso los retorna al pool de libres ejecutando la operación *release*. Los parámetros a *release* indican la identidad de las unidades que son retornadas. Ignorando la representación de las unidades del recurso, las operaciones *request* y *release* tienen la siguiente forma general:

request(parámetros): ⟨ **await** request puede ser satisfecho → tomar unidades ⟩

release(parámetros): ⟨ retornar unidades ⟩

Las operaciones necesitan ser atómicas dado que ambas necesitan acceder a la representación de las unidades del recurso. Siempre que esta representación use variables diferentes de otras variables del programa, las operaciones aparecerán como atómicas con respecto a otras acciones y por lo tanto pueden ejecutar concurrentemente con otras acciones.

Este patrón de solución general puede ser implementado usando la técnica de passing the baton. En particular, *request* tiene la forma de F_2 de modo que es implementada por un fragmento similar a (4.7):

(4.9) *request*(parámetros): **P**(e)
if request no puede ser satisfecho → DELAY **fi**
toma unidades
SIGNAL

Similarmente, *release* la forma de F_1 de modo que es implementada por un fragmento de programa similar a (4.6):

(4.10) *release*(parámetros): **P**(e)
retorna unidades
SIGNAL

Como antes, *e* es un semáforo que controla la entrada a las operaciones, y SIGNAL es un fragmento de código como (4.8); SIGNAL o despierta un proceso demorado (si algún pedido demorado puede ser satisfecho) o ejecuta **V**(e). El código de DELAY es un fragmento similar al mostrado en (4.7): registra que hay un request demorado, ejecuta **V**(e), luego se demora en un semáforo de condición. Los detalles exactos de cómo es implementado SIGNAL para un problema específico depende de cómo son las condiciones de demora diferentes y cómo son representadas. En cualquier evento, el código DELAY necesita salvar los parámetros que describen un pedido demorado de modo que puedan ser examinados en SIGNAL. Además, se necesita que haya un semáforo de condición por cada condición de demora diferente.

La próxima sección desarrolla una solución a un problema de alocación de recursos específica. La solución ilustra cómo resolver tal problema.

Alocación Shortest-Job-Next

Es una política de alocación usada para distintas clases de recursos. Asumimos que los recursos compartidos tienen una única unidad (luego consideraremos el caso general). Esta política se define como sigue:

(4.11) **Alocación Shortest-Job-Next (SJN)**. Varios procesos compiten por el uso de un único recurso compartido. Un proceso requiere el uso del recurso ejecutando *request*(*time*,*id*), donde *time* es un entero que especifica cuánto va a usar el recurso el proceso, e *id* es un entero que identifica al proceso que pide. Cuando un proceso ejecuta *request*, si el recurso está libre, es inmediatamente alocado al proceso; si no, el proceso se demora. Después de usar el recurso, un proceso lo libera ejecutando *release*( ). Cuando el recurso es liberado, es alocado al proceso demorado (si lo hay) que tiene el mínimo valor de *time*. Si dos o más procesos tienen el mismo valor de *time*, el recurso es alocado al que ha esperado más.

Por ejemplo, la política SJN puede ser usada para alocación de procesador (en la cual *time* es el tiempo de ejecución), para spooling de archivos a una impresora (*time* sería el tiempo de impresión), o para servicio de file trasfer (ftp) remoto (*time* sería el tiempo estimado de transferencia). La política SJN es atractiva pues minimiza el tiempo promedio de ejecución. Sin embargo, es inherentemente unfair: un proceso puede ser demorado para siempre si hay una corriente continua de requests especificando tiempos de uso menores. (Tal unfairness es extremadamente improbable en la práctica a menos que el recurso esté totalmente sobrecargado. Si interesa unfairness, la política SJN puede ser levemente modificada de modo que un proceso que ha estado demorado un largo tiempo tenga preferencia; esta técnica es llamada “aging”).

Si un proceso hace un pedido y el recurso está libre, el pedido puede ser satisfecho inmediatamente pues no hay otros pedidos pendientes. Así, el aspecto SJN de la política de alocación se pone en juego solo si más de un proceso tiene un pedido pendiente. Dado que hay un solo recurso, es suficiente usar una variable booleana para registrar si el recurso está disponible. Sea *free* una variable booleana que es true cuando el recurso está disponible y false cuando está en uso. Para implementar la política SJN, los request pendientes necesitan ser recordados y ordenados. Sea *P* un conjunto de pares (*time*, *id*), ordenado por los valores del campo *time*. Si dos pares tienen el mismo valor para *time*, están en P en el orden en el que fueron insertados. Con esta especificación, el siguiente predicado debe ser un invariante global:

SJN: *P* es un conjunto ordenado $\wedge$ *free* $\Rightarrow$ (*P* = $\varnothing$)

En síntesis, *P* está ordenado, y, si el recurso está libre, *P* es el conjunto vacío. Inicialmente, *free* es true y *P* es vacío, de modo que SJN es trivialmente true.

Ignorando la política SJN por el momento, un request puede ser satisfecho exactamente cuando el recurso está disponible. Esto resulta en la solución coarse-grained:

var free := true

request(time,id): ⟨ **await** free $\rightarrow$ free := false ⟩

release( ): ⟨ free := true ⟩

Sin embargo, con la política SJN un proceso que ejecuta *request* necesita demorarse hasta que el recurso esté libre y el pedido del proceso es el próximo en ser atendido de acuerdo a la política SJN. A partir del segundo conjuntor de SJN, si *free* es true en el momento en que un proceso ejecuta *request*, el conjunto P está vacío. Por lo tanto la condición de demora es suficiente para determinar si un pedido puede ser satisfecho inmediatamente. El parámetro *time* incide solo si un request debe ser demorado, es decir, si *free* es false. Basado en estas observaciones, podemos implementar *request* como se mostró en (4.9):

```
request(time, id):  P(e)
                    if not free → DELAY fi
                    free := false
                    SIGNAL
```

Y podemos implementar *release* como mostraba (4.10):

```
release( ): P(e)
            free := true
            SIGNAL
```

En *request*, asumimos que las operaciones **P** sobre el semáforo de entrada *e* se completan en el orden en el cual fueron intentados; es decir, **P**(e) es FCFS. Si esto no es así, los pedidos no necesariamente serán servidos en orden SJN.

Lo que resta es implementar el aspecto SJN de la política de alocación. Esto involucra usar el conjunto P y semáforos para implementar DELAY y SIGNAL. Cuando un request no puede ser satisfecho, necesita ser salvado para ser examinado más tarde cuando el recurso es liberado. Así, en DELAY un proceso necesita:

- insertar sus parámetros en P
- liberar el control de la SC protegiendo *request* y *release* ejecutando **V**(e), luego
- demorarse en un semáforo

Cuando el recurso es liberado, si el conjunto P no está vacío, el recurso necesita ser alocado a exactamente un proceso de acuerdo con la política SJN. En este caso, insertar los parámetros en P corresponde a incrementar un contador c_i en (4.7). En ambos casos, un proceso indica que está a punto de demorarse e indica qué condición está esperando.

En anteriores ejemplos (como la solución a lectores/escritores) había solo unas pocas condiciones de demora distintas, y por lo tanto se necesitaban pocos semáforos de condición. Aquí, sin embargo, cada proceso tiene una condición de demora distinta, dependiendo de su posición en P: el primer proceso en P necesita ser despertado antes del segundo, y así siguiendo. Asumimos que hay *n* procesos que usan el recurso. Sea *b*[1:n] un arreglo de semáforos, donde cada entry es inicialmente 0. También asumimos que los valores de *id* son únicos y están en el rango de 1 a *n*. Entonces el proceso *id* se demora sobre el semáforo *b*[id]. Aumentando *request* y *release* con los usos de P y *b* como especificamos, tenemos la siguiente solución al problema de alocación SJN:

```
var free := true, e : sem := 1, b[1:n] : sem := ( [n] 0 )
var P : set of (int, int) := ∅
{ SJN: P es un conjunto ordenado ∧ free ⇒ ( P = ∅ ) }
request(time, id):  P(e)        {SJN}
                    if not free → insert(time,id) en P; V(e); P(b[id]) fi
                    free := false       {SJN}
                    V(e)    # optimizado pues free es false en este punto
release( ): P(e)          {SJN}
            free := true
            if P ≠ ∅ → remover el primer par (time,id) de P; V(b[id])
            □ P = ∅ → {SJN}  V(e)
```

fi

En esta solución, el insert en *request* se asume que pone el par en el lugar apropiado en P para mantener el primer conjunto de SJN. Luego, SJN es invariante fuera de *request* y *release*; es decir, SJN es true luego de cada **P**(e) y antes de cada **V**(e). La primera sentencia guardada en el código de señalización en *release* despierta exactamente un proceso si hay un pedido pendiente, y por lo tanto P no está vacío. El "baton" es pasado a ese proceso, el cual setea *free* a false. Esto asegura que el segundo conjuntor en SJN es true si P no es vacío. Dado que hay un único recurso, no se pueden satisfacer requests posteriores, de modo que el código signal en *request* es simplemente **V**(e).

Los semáforos b[id] son ejemplos de lo que se llaman semáforos privados.

(4.12) **Semáforo Privado**. El semáforo *s* es llamado semáforo privado si exactamente un proceso ejecuta operaciones **P** sobre *s*.

Los semáforos privados son útiles en cualquier situación en la cual es necesario ser capaz de señalar procesos individuales. Para algunos problemas de alocación, sin embargo, puede haber menos condiciones de demora diferentes que procesos que compiten por un recurso. En ese caso, puede ser más eficiente usar un semáforo para cada condición diferente que usar semáforos privados para cada proceso. Por ejemplo, si los bloques de memoria son alocados solo en unos pocos tamaños (y no interesa en qué orden los bloques son alocados a los procesos que compiten por el mismo tamaño) entonces sería suficiente tener un semáforo de demora para cada tamaño diferente.

Podemos generalizar rápidamente la solución a recursos con más de una unidad. En este caso, cada unidad estaría libre o alocado, y *request* y *release* tendrían un parámetro, *amount*, para indicar cuántas unidades requiere o devuelve un proceso. Podríamos modificar la solución como sigue:

- Reemplazar *free* por un entero *avail* que registra el número de unidades disponibles
- En *request*, testear si *amount* unidades están libres, es decir, si *amount* ≤ *avail*. Si es así, alocarlas; si no, registrar cuántas unidades se requieren antes de demorarse
- En *release*, incrementar *avail* por *amount*, luego determinar si el proceso demorado hace más tiempo que tiene el mínimo valor para *time* puede ser satisfecho. Si es así, despertarlo; si no, ejecutar **V**(e).

La otra modificación es que ahora podría ser posible satisfacer más de un request pendiente cuando las unidades son liberadas. Por ejemplo, podría haber dos procesos demorados que requieren juntos menos unidades que las que fueron liberadas. En este caso, el que es despertado primero necesita hacer signal al segundo después de tomar las unidades que requiere. En resumen, el protocolo de signal al final de *request* necesita ser el mismo que el del final de *release*.

IMPLEMENTACION

Dado que las operaciones sobre semáforos son casos especiales de sentencias **await**, podemos implementarlas usando busy waiting y la técnica del capítulo 3. Sin embargo, la única razón por la que uno querría hacerlo así es para ser capaz de escribir programas usando semáforos más que spin locks y flags de bajo nivel. En consecuencia, mostraremos cómo agregar semáforos al kernel descripto en el capítulo 3. Esto involucra aumentar el kernel con descriptores de semáforo y tres primitivas adicionales: *create_sem*, *P*, y *V*.

Un descriptor de semáforo contiene el valor de un semáforo; es inicializado invocando *create_sem*. Las primitivas *P* y *V* implementan las operaciones **P** y **V**. Asumimos aquí que todos los semáforos son generales; por lo tanto la operación **V** nunca bloquea. En esta sección, describimos cómo estas componentes son incluidas en un kernel monoprocesador y cómo cambiar el kernel resultante para soportar múltiples procesadores.

Recordemos que en el kernel monoprocesador, un proceso a la vez estaba ejecutando, y todos los otros estaban listos para ejecutar. Como antes, el índice del descriptor del proceso ejecutante es almacenado en la variable *executing*, y los descriptores para todos los procesos listos son almacenados en la ready list. Cuando se agregan semáforos, hay un tercer estado de proceso posible: bloqueado. En particular, un proceso está bloqueado si está esperando completar una operación **P**. Para seguir la pista de los procesos bloqueados, cada descriptor de semáforo contiene una lista enlazada de los descriptores de procesos bloqueados en ese semáforo. En un monoprocesador, exactamente un proceso está ejecutando, y su descriptor no está en ninguna lista; todo otro descriptor de proceso o está en la ready list o en la lista de bloqueados de algún semáforo.

Para cada declaración de semáforo en un programa concurrente, se genera un llamado a la primitiva *create_sem*; el valor inicial del semáforo se pasa como argumento. La primitiva *create_sem* encuentra un descriptor de semáforo vacío, inicializa el valor y la lista de bloqueados, y retorna un "nombre" para el descriptor. Este nombre típicamente es la dirección del descriptor o un índice en una tabla que contiene la dirección.

Después de que un semáforo es creado, es usado para invocar las primitivas **P** y **V**. Ambas tienen un único argumento que es el nombre de un descriptor de semáforo. La primitiva P chequea el valor en el descriptor. Si el valor es positivo, es decrementado; en otro caso, el descriptor del proceso ejecutante se inserta en la lista de bloqueados del semáforo. Similarmente, V chequea la lista de bloqueados del descriptor del semáforo. Si está vacía, el valor del semáforo es imcrementado; en otro caso un descriptor de proceso es removido de la lista de bloqueados e insertado en la ready list, Es común para cada lista de bloqueados implementarla como una cola FIFO pues asegura que las operaciones sobre semáforos son fair.

Las outlines de estas primitivas son las siguientes (se agregan a las rutinas en el kernel monoprocesador ya visto. Nuevamente, el procedure *dispatcher* es llamado al final de cada primitiva; sus acciones son las mismas que antes):

```
procedure create_sem(valor inicial) returns name
  tomar un descriptor de semáforo vacío
  inicializar el descriptor
  setear name con el nombre (índice) del descriptor
  call dispatcher( )
end
procedure P(name)
  encontrar el descriptor de semáforo de name
  if valor > 0 → valor := valor - 1
  □ valor = 0 → insertar descriptor de executing al final de la lista de
bloqueados
                executing := 0     # indica que executing ahora está bloqueado
  fi
  call dispatcher( )
end
procedure V(name)
  encontrar el descriptor de semáforo de name
  if lista de bloqueados vacía → valor := valor + 1
  □ lista de bloqueados no vacía →
        remover el descriptor de proceso del principio de la lista de bloqueados
        insertar el descriptor al final de la ready list
  fi
  call dispatcher( )
end
```

Por simplicidad, la implementación de las primitivas de semáforo no reusan descriptores de semáforos. Esto sería suficiente si todos los semáforos son globales a los procesos, pero en general esto no sucede. Así usualmente es necesario reusar descriptores de semáforos como los descriptores de proceso. Una aproximación es que el kernel provea una primitiva adicional *destroy_sem*; sería invocada por un proceso cuando ya no necesita un semáforo. Una alternativa es registrar en el descriptor de cada proceso los nombres de todos los semáforos que ese proceso creó. Luego, los semáforos que creó podrían ser destruidos por el kernel cuando el proceso invoca la primitiva *quit*. Con esto, es imperativo que un semáforo no sea usado luego de ser destruido.

Podemos extender la implementación monoprocesador a una para un multiprocesador de la misma manera que en el capítulo 3. Nuevamente, el requerimiento crítico es bloquear las estructuras de datos compartidas, pero solo por el mínimo tiempo requerido. Por lo tanto, debería haber un lock separado para cada descriptor de semáforo. Este descriptor es lockeado en P y V justo antes de ser accedido; el lock es liberado tan pronto como el descriptor ya no se necesita. Como en el primer kernel multiprocesador, los locks son adquiridos y liberados por una solución busy-waiting al problema de la SC.

Regiones Críticas Condicionales

Los semáforos son un mecanismo de sincronización fundamental. Como vimos, pueden ser usados sistemáticamente para resolver cualquier problema de sincronización. Sin embargo, resolver un problema de exclusión mutua requiere encontrar el cambio de variables justo para usar. Además, resolver un problema de sincronización por condición requiere usar varios semáforos y variables adicionales en un protocolo relativamente complejo.

Una dificultad peor con los semáforos es que la naturaleza de bajo nivel de **P** y **V** hace que sea bastante fácil equivocarse al usarlos. El programador debe tener cuidado de no omitir accidentalmente un **P** o un **V**, de no ejecutar demasiadas operaciones **P** o **V**, de emplear un semáforo equivocado, o de fallar al proteger todas las SC.

Un problema final con semáforos es que uno programa tanto exclusión mutua como sincronización por condición usando el mismo par de primitivas. Esto hace difícil identificar el propósito de un **P** o **V** sin mirar las otras operaciones sobre el mismo semáforo. Dado que la exclusión mutua y la sincronización por condición son conceptos distintos, deberían ser programados de distintas maneras.

Las *regiones críticas condicionales* (CCR) solucionan estas dificultades proveyendo una notación estructurada para especificar sincronización. Con CCRs, las variables compartidas que necesitan ser accedidas con exclusión mutua son declaradas en recursos. Las variables en un recurso son accedidas solo en sentencias **region** que nombran el recurso. La exclusión mutua es provista garantizando que la ejecución de sentencias **region** que nombran el mismo recurso no es interleaved. La sincronización por condición se provee por condiciones booleanas en las sentencias **region**.

Con CCRs, la exclusión mutua es implícita, y la sincronización por condición es programada explícitamente. Esto hace a las CCRs generalmente más fáciles de usar que los semáforos. También lleva a un sistema de prueba más simple. Sin embargo, las sentencias **region** son más “caras” de implementar que las operaciones sobre semáforos. Esto es porque las condiciones de demora en las sentencias **region** tienen que ser reevaluadas cada vez que cambia una variable compartida o cada vez que un kernel necesita decidir si algún proceso bloqueado puede ejecutar.

Dado que las CCRs son relativamente ineficientes, no fueron tan usadas como otros mecanismos. Pero, todavía son estudiadas por algunas razones:

- Son el paso histórico de semáforos a monitores
- Ilustran cómo el uso de sentencias de sincronización estructuradas pueden eliminar interferencia
- Introdujeron el uso de condiciones de sincronización booleanas, la cual se usó luego en varias notaciones de lenguajes

El resto del capítulo define la sintaxis y semántica de CCRs y da varios ejemplos que ilustran el uso. Los ejemplos ilustran el poder expresivo de CCRs en relación a los semáforos.

NOTACION Y SEMANTICA

La notación para CCRs impone restricciones al compilador en el uso de variables compartidas. Estas restricciones hacen a los programas más estructurados y llevan a un sistema de prueba más simple en el cual la interferencia es evitada automáticamente.

La notación de CCRs emplea dos mecanismos: declaraciones **resource** y sentencias **region**. Un **resource** es una colección nombrada de variables compartidas a la cual se requiere acceso mutuamente exclusivo. La forma de una declaración es:

resource *r* (declaraciones de variables)

El identificador *r* es el nombre del recurso; el nombre puede ser un identificador simple o subindizado. Las componentes de *r* son una o más declaraciones de variables. Son declaradas como campos de registro omitiendo la palabra clave **var**.

Cada variable compartida en un programa debe pertenecer a un recurso. Las variables dentro de un recurso pueden ser accedidas *solo* dentro de sentencias **region** que nombran el recurso. Una sentencia **region** tiene la forma:

region *r* **when** $B \rightarrow S$ **end**

donde *r* es el nombre del recurso, *B* es una guarda booleana, y *S* es una lista de sentencias. Tanto *B* como *S* pueden referenciar las variables de *r* así como variables locales al proceso ejecutante. La frase **when** es opcional; puede omitirse si *B* no es necesaria.

La ejecución de una sentencia **region** demora el proceso ejecutante hasta que *B* es true; luego se ejecuta *S*. La ejecución de sentencias **region** que nombran el mismo recurso es mutuamente exclusiva; en particular, *B* se garantiza que es true cuando comienza la ejecución de *S*. Así, una sentencia **region** es muy similar a un **await**. Las diferencias son que una sentencia **region** explícitamente nombra la colección de variables compartidas que accederá y la condición de demora puede depender solo de estas variables. Como veremos, esto hace a la semántica de las CCRs más simple que la semántica de los programas que usan sentencias **await**; además hace a las sentencias **region** más simples de implementar que las sentencias **await** arbitrarias.

Dos sentencias **region** pueden ser anidadas si nombran distintos recursos. Cuando se ejecuta la sentencia **region** anidada, un proceso tiene acceso exclusivo a las variables de ambos recursos. Además, un proceso retiene el acceso exclusivo al recurso externo aún si el proceso es demorado esperando ejecutar la sentencia **region** anidada. Así, podría ocurrir deadlock si dos procesos anidan sentencias **region** en orden distinto.

Un compilador para un lenguaje que contiene CCRs puede chequear que todas las variables compartidas estén dentro de recursos y que son accedidas solo dentro de sentencias **region**. Así un compilador puede asegurar que las variables compartidas son accedidas con exclusión mutua. Por contraste, tales chequeos en tiempo de compilación no pueden ser hechos para programas que emplean semáforos.

Ejemplos

Antes de definir la semántica formal de CCRs, damos dos ejemplos básicos para ilustrar cómo son usadas e interpretadas. El primer ejemplo muestra cómo simular semáforos usando CCRs.

Sea *s* un semáforo (general) con valor inicial $n \geq 0$. Usando CCRs, un semáforo con valor inicial *n* es representado por un recurso:

resource *sem* (*s* : **int** := *n*)

Las operaciones **P** y **V** sobre *s* son simuladas por:

P(*s*): **region** *sem* **when** $s > 0 \rightarrow s := s - 1$ **end**
V(*s*): **region** *sem* $\rightarrow s := s + 1$ **end**

Nótese la similitud entre esta implementación y la anterior en términos de sentencias **await**.

Como segundo ejemplo (que ilustra mejor cómo difieren los semáforos y CCRs) consideremos cómo implementar un buffer de un solo slot. Recordemos que tal buffer tiene un único slot que puede contener un mensaje y que el slot es accedido por dos operaciones: *deposit* y *fetch*. Como antes, la ejecución de *deposit* y *fetch* deben alternarse. Para representar el buffer se necesitan dos variables: una para el buffer en si mismo y otra para sincronizar la ejecución de *deposit* y *fetch*. Para sincronización, es suficiente usar una variable booleana que indica si el buffer está lleno o vacío. El buffer es representado por el recurso

resource *buffer* (*b* : T; *full* : **bool** := false)

donde T es algún tipo de datos. Dada esta representación, *deposit* y *fetch* se programan como:

deposit: **region** *buffer* **when not** *full* $\rightarrow$ *b* := data; *full* := true **end**
fetch: **region** *buffer* **when** *full* $\rightarrow$ result := *b*; *full* := false **end**

Aquí tanto la sincronización como el acceso al buffer son combinados en una sentencia **region** única. Nótese cómo difiere esta implementación del capítulo 4 en la cual las sentencias **await** eran usadas para implementar solo sincronización. En general, CCRs llevan a soluciones más compartas a problemas de sincronización pues se requieren menos sentencias. Además, las sentencias **region** permiten que las condiciones de sincronización sean expresadas directamente como expresiones booleanas; con semáforos, las condiciones de sincronización son implícitas.

Reglas de Inferencia

La estructura provista por CCRs (específicamente que todas las variables compartidas deben pertenecer a recursos y que las variables recurso sólo pueden ser accedidas por sentencias **region**) lleva a un sistema de prueba que es más simple que el introducido en el cap. 2. La principal simplificación es que la interferencia se evita automáticamente, y por lo tanto no es necesario probar no interferencia.

El orden en el cual los procesos ejecutan sentencias **region** que nombran el mismo recurso es no determinístico si las guardas en ambas sentencias son true. Así, para especificar los estados aceptables de las variables del recurso, es necesario usar un predicado que es un invariante global: debe ser true inicialmente y debe ser true antes y después de cada sentencia **region**. Con CCRs, el predicado asociado con un recurso es llamado *resource invariant*. Hay uno de tales invariantes por cada recurso.

Para evitar interferencia, todas las variables referenciadas en un resource invariant deben pertenecer al recurso asociado. Para relacionar los estados de distintos procesos con los otros, usualmente son necesarias variables auxiliares. Deben ser agregadas a los recursos e incluidas en resource invariants. Como las variables de programa, tales variables deben ser referenciadas y alteradas solo dentro de sentencias **region**. Una variable auxiliar también puede aparecer en aserciones en la prueba de un proceso, siempre que la variable no sea cambiada por otro proceso. Esto permite que los estados de procesos sean relacionados sin introducir interferencia potencial.

Sea *r* un recurso, y sea *RI* el resource invariant asociado. Asumimos que *RI* es inicialmente true. Luego, cualquier sentencia **region**

region r **when** *B* $\rightarrow$ *S* **end**

puede asumir que tanto *RI* como *B* son true cuando la ejecución de *S* comienza así como la ejecución de *S* reestablece *RI*. Esto lleva a la siguiente regla de inferencia para sentencias **region**:

(5.1) **Regla de Region**: $\{ P \wedge RI \wedge B \}\ S\ \{ Q \wedge RI \}$
ninguna variable libre en P o Q
es cambiada en otro proceso

$\{ P \}$ **region** r **when** $B \rightarrow S$ **end** $\{ Q \}$

Como es usual, P y Q son aserciones locales al proceso que ejecuta una sentencia **region**. Estas aserciones pueden referenciar variables compartidas que no son cambiadas por otros procesos, pero solo RI puede referenciar variables compartidas que podrían ser cambiadas por otros procesos.

La Regla de Region es similar a la Regla de Sincronización (2.7). La diferencia es que el resource invariant RI explícitamente aparece en la hipótesis pero no aparece en la conclusión. Así, los resource invariants *no* aparecen en las pruebas de procesos individuales. Los resource invariants son introducidos en la prueba de un programa empleando una regla de inferencia distinta para la sentencia **co**.

Supongamos que un programa concurrente contiene *m* recursos, con invariantes asociados RI_1, ..., RI_m. Asumimos que los procesos sincronizan solo por medio de sentencias **region**. También que todas las variables compartidas pertenecen a recursos y son accedidas solo dentro de sentencias **region**. Entonces cuando usamos CCRs, la Regla de Concurrencia (2.11) es reemplazada por:

(5.1) **Regla de CCRs**:

$$\frac{\begin{array}{c}\{ P_i \}\ S_i\ \{ Q_i \},\ 1 \le i \le n \\ \text{ninguna variable libre en } P_i \text{ o } Q_i \text{ es cambiada en } S_j,\ i \ne j \\ \text{todas las variables en } RI_k \text{ son locales al recurso } r_k,\ 1 \le k \le m\end{array}}{\begin{array}{l}\{ RI_1 \wedge ... \wedge RI_m \wedge P_1 \wedge ... \wedge P_n \} \\ \textbf{co}\ S_1\ //\ ....\ //\ S_n\ \textbf{oc} \\ \{ RI_1 \wedge ... \wedge RI_m \wedge Q_1 \wedge ... \wedge Q_n \}\end{array}}$$

Esta regla de inferencia requiere que los resource invariants sean inicialmente true. Luego (como resultado de la regla de region) los resource invariants siempre serán true, excepto quizás cuando una sentencia **region** está siendo ejecutada. Dado que las **region** son mutuamente exclusivas por definición, los resource invariants no pueden ser interferidos. Además, las aserciones en los procesos no pueden referenciar variables cambiadas por otros procesos. Por estas razones, los procesos no pueden interferir.

La ausencia de la necesidad de probar no interferencia es la diferencia importante entre la Regla de CCRs y la Regla de Concurrencia. Resulta del requerimiento al compilador de que las variables compartidas pertenezcan todas a recursos accedidos solo por sentencias **region** mutuamente exclusivas.

PROBLEMA DE LOS FILOSOFOS REVISITADO

Con CCRs, las soluciones a problemas de sincronización son derivadas en forma similar a como lo hacíamos. Los primeros tres pasos son idénticos: especificar un invariante global, hacer un outline de solución, luego guardar las asignaciones a variables compartidas para asegurar que el invariante es true. Nuevamente, programaremos soluciones coarse-grained usando sentencias **await**:

El último paso de derivación es implementar las sentencias **await**. Con CCRs, esto es directo pues las sentencias **region** son muy similares a las sentencias **await**. Las variables compartidas que son usadas juntas son ubicadas en el mismo recurso, y las sentencias **await** son reemplazadas por sentencias **region**. Dado que las sentencias **region** proveen directamente exclusión mutua y sincronización por condición, no se requieren pasos intrincados que usan sincronización fine-grained o semáforos.

Esta aproximación puede usarse siempre. La única complicación es que las variables en el mismo recurso son accedidas siempre con exclusión mutua, y esto puede restringir posible concurrencia. Obtener máxima concurrencia requiere usar más recursos o remover algunas variables compartidas de los recursos. Sin embargo, hacer esto puede hacer más difícil asegurar propiedades de sincronización requeridas.

Esta sección deriva una nueva solución al problema de los filósofos. La solución también ilustra varios aspectos de la semántica formal de CCRs.

Recordemos que para comer, un filósofo necesita adquirir dos tenedores, uno en cada lado. En el cap. 4, el requerimiento de sincronización fue especificado en términos de los estados de los tenedores: Cada tenedor podría ser levantado a lo sumo una vez más que las que fue bajado. Esta especificación llevó a la solución basada en semáforos. Aquí, expresaremos la sincronización en términos del estado de los filósofos. Cualquier especificación es aceptable, pero se obtienen soluciones distintas.

Como antes, simulamos las acciones de los filósofos como sigue:

```
Philosopher[i:1..5]:: do true → piensa
                                adquiere tenedores
                                come
                                libera tenedores
                      od
```

Un estado malo para este programa es uno en el cual filósofos vecinos están comiendo al mismo tiempo. Sea *eating*[i] true si Philosopher[i] está comiendo, y falso en otro caso. Entonces los estados buenos del programa están caracterizados por:

EAT: ($\forall$ i : 1 $\leq$ i $\leq$ 5 : eating[i] $\Rightarrow$ $\neg$ (eating[iΘ1] $\vee$ eating[i$\oplus$1]))

En EAT, Θ denota el vecino izquierdo, y $\oplus$ denota el vecino derecho.

Adquirir los tenedores involucra setear *eating*[i] a true; liberar los tenedores involucra setear *eating*[i] a false. Para asegurar que EAT es un invariante, necesitamos guardar el seteo *eating*[i] a true; la guarda apropiada es el consecuencia de la implicación en EAT. Sin embargo, no necesitamos guardar el seteo a false pues esa acción hace falso el antecedente de una implicación.

La siguiente es una solución coarse-grained resultante de esta especificación (se incluyen aserciones sobre el valor de *eating* para cada filósofos en puntos apropiados. Recordemos que esto está permitido pues *eating*[i] solo es cambiada por Philosopher[i]):

```
var eating[1:5] : bool := ( [5] false )
EAT: ( ∀ i : 1 ≤ i ≤ 5 : eating[i] ⇒ ¬ ( eating[iΘ1] ∨ eating[i⊕1] ) )
Philosopher[i:1..5]::
  do true →      { ¬ eating[i] }
     piensa
     ⟨ await not (eating[iΘ1] or eating[i⊕1]) → eating[i] := true ⟩
     { eating[i] }
     come
     ⟨ eating[i] := false ⟩
  od
```

El programa para cada filósofo está ahora en una forma que puede ser convertida directamente en una que usa CCRs. El arreglo *eating* se convierte en un recurso, y las sentencias que referencian los valores de *eating* se convierten en sentencias **region**. Haciendo estos cambios tenemos la solución final:

```
resource table(eating[1:5] : bool := ( [5] false ) )
EAT: ( ∀ i : 1 ≤ i ≤ 5 : eating[i] ⇒ ¬ ( eating[iΘ1] ∨ eating[i⊕1] ) )
Philosopher[i:1..5]::
  do true →      { ¬ eating[i] }
     piensa
     region table when not (eating[iΘ1] or eating[i⊕1]) →eating[i] := true end
     { eating[i] }
     come
     region table → eating[i] := false end
  od
```

Aquí, todas las variables compartidas están en el único recurso *table*, y el resource invariant referencia solo estas variables. Las aserciones se desprenden de la inicialización de *eating* y de las aplicaciones de la Regla de Region (5.1). Más aún, como notamos, las aserciones en cada proceso satisfacen el requerimiento de que no referencian variables cambiadas por otro proceso. Así, la proof outline satisface la hipótesis de la Regla de CCRs (5.2).

Dado que construimos la solución para asegurar la invarianza de EAT, tiene la propiedad de exclusión requerida de que los filósofos vecinos no pueden comer al mismo tiempo. La solución además está libre de deadlock. Informalmente, esto es porque si todos los procesos están demorados en sus primeras sentencias **region**, entonces todos los valores de *eating* son falsos; por lo tanto al menos un filósofo puede continuar.

Sin embargo, la solución no es fair aún con una política de scheduling fuertemente fair. Esto es porque un filósofo podría ser bloqueado y sus vecinos podrían "conspirar" de modo que uno o el otro esté siempre comiendo. Aunque cada vecino podría periódicamente no estar comiendo, el estado en el cual ninguno está comiendo podría no ocurrir nunca. Esto es altamente indeseable (asumiendo que los filósofos son de hecho "pensadores") pero es teóricamente posible.

LECTORES/ESCRITORES REVISITADO

En el capítulo 4, introdujimos el problema de lectores/escritores y desarrollamos dos soluciones usando semáforos para sincronización. Esas soluciones podrían emplear CCRs simulando las operaciones semáforo. Sin embargo, podemos usar CCRs para obtener soluciones mucho más compactas. Esta sección desarrolla dos soluciones: la primera da preferencia a los lectores; la segunda da preferencia a los escritores e ilustra el uso de dos sentencias **region** en el protocolo de entrada.

Solución de Preferencia de Lectores

Recordemos la definición del problema de lectores/escritores dada en (4.5): los procesos lectores examinan una BD compartida, y los escritores la examinan y la alteran. Los lectores pueden acceder la BD concurrentemente, pero los escritores requieren acceso exclusivo. Siguiendo la especificación dada en el cap. 4, sea *nr* el número de lectores accediendo la BD, y sea *nw* el número de escritores. Entonces la sincronización requerida es especificada por:

$$RW: (nr = 0 \lor nw = 0) \land nw \leq 1$$

Modelizando las acciones de los procesos por loops perpetuos como antes, esta especificación da la solución coarse-grained vista en el cap. 4:

```
var nr := 0, nw := 0
{ RW: ( nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1 }
Reader[i:1..m] :: do true → ⟨ await  nw = 0 → nr := nr + 1 ⟩
                              lee la BD
                              ⟨ nr := nr - 1 ⟩
                  od
Writer[j:1..n] :: do true → ⟨ await  nr = 0 and nw = 0 → nw := nw + 1 ⟩
                            escribe la BD
                            ⟨ nw := nw - 1 ⟩
                  od
```

Podemos convertir directamente esta solución en un programa que usa CCRs. Nuevamente ubicamos las variables compartidas en un recurso y reemplazamos cada sentencia **await** por una sentencia **region**. Haciendo estas sustituciones obtenemos el siguiente programa (la solución es mucho más compacta que cualquiera de las soluciones basadas en semáforos. Esto resulta de la exclusión implícita de las sentencias **region** y el uso de guardas booleanas para especificar condiciones de sincronización):

```
resource rw(nr := 0, nw := 0)
{ RW: ( nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1 }
Reader[i:1..m] :: do true → region rw when nw = 0 → nr := nr + 1 end
                                lee la BD
                                region rw → nr := nr - 1 end
                  od
Writer[j:1..n] :: do true → region rw when nr = 0 and nw = 0 →nw :=nw+1 end
                                escribe la BD
                                region rw → nw := nw - 1 end
                  od
```

Dado que esta es solo una recodificación de una solución correcta, es correcta. Sin embargo, el programa no contiene la información suficiente para probar formalmente que la solución tiene la exclusión requerida y está libre de deadlock. Por ejemplo, para probar que dos escritores no pueden acceder simultáneamente la BD, es necesario saber que ambos están tratando de hacerlo, y luego establecer una contradicción. Formalmente probar exclusión mutua y ausencia de deadlock requiere agregar variables auxiliares y relacionar sus valores con los valores de *nr* y *nw*.

Solución de Preferencia de Escritores

La solución anterior da preferencia a los lectores: Si algún lector está en la SC, otro lector puede entrar a su SC, pero un escritor es demorado. Esencialmente esto es porque la condición de entrada para los escritores es más fuerte que para los lectores: Un escritor tiene que esperar hasta que no haya lectores ni escritores, mientras un lector solo tiene que esperar que no haya escritores. Para dar preferencia a los escritores, podríamos usar CCRs para simular el método semáforo de passing the baton. Sin embargo, podemos usar CCRs directamente para implementar una solución de preferencia de escritores. La idea básica es fortalecer la condición de entrada para los lectores.

Recordemos que una solución que da preferencia a los escritores es una que asegura que los escritores toman acceso a la BD tan pronto como sea posible. En particular, un lector que quiere acceder a la BD es demorado si un escritor ya está accediendo la BD *o* si algún escritor está demorado. Para expresar la propiedad de preferencia, sea *ww* el número de escritores esperando. Entonces si *ww* es positivo, nuevos lectores no pueden comenzar a leer la BD, es decir que *nr* no puede ser incrementada. Esto es especificado por:

$$ww \geq 0 \wedge (ww > 0 \Rightarrow \text{nr no puede crecer })$$

Desafortunadamente, el segundo conjuntor no puede ser expresado fácilmente en lógica de predicados. Esto es porque expresa una relación entre pares de estados más que una propiedad que es true de un único estado. Sin embargo puede ser usado como una especificación informal pues especifica una propiedad de seguridad, siendo lo malo una transición de estado en la cual *ww* es positivo pero *nr* es incrementado.

Para obtener una solución que da preferencia a los escritores, necesitamos modificar la solución anterior para incluir el requerimiento adicional especificado arriba. En particular, agregamos *ww* al recurso *rw*. Los escritores incrementan *ww* para indicar que quieren acceder a la BD; un escritor decrementa *ww* una vez que adquirió el acceso. Finalmente, la guarda en el protocolo de entrada del lector es fortalecida para incluir el conjuntor *ww* = 0; esto asegura que *nr* no es incrementada si *ww* > 0.

La siguiente solución incorpora estos cambios:

```
resource rw(nr := 0, nw := 0, ww := 0)
{ RW: ( nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1 }
Reader[i:1..m] :: do true → region rw when nw = 0 and ww = 0 →
                                    nr := nr + 1 end
                                lee la BD
                                region rw → nr := nr - 1 end
                  od
Writer[j:1..n] :: do true → region rw → ww := ww + 1 end
```

```
                    region rw when nr = 0 and nw = 0 →
                        nw :=nw+1; ww := ww - 1 end
                    escribe la BD
                    region rw → nw := nw - 1 end
            od
```

La solución usa dos sentencias **region**, una después de la otra, en el entry protocol para los escritores. La primera registra que arribó un escritor. La segunda se usa para demorar hasta que sea seguro acceder la BD. Esta técnica suele usarse para resolver problemas de scheduling usando CCRs.

COMUNICACION INTERPROCESO

Recordemos que un buffer limitado es un buffer multislot usado para comunicación entre procesos productores y consumidores. En particular, un buffer limitado es una cola FIFO accedida por operaciones *deposit* y *fetch*. Los productores ponen mensajes en el buffer ejecutando *deposit*, demorando si es necesario hasta que haya un slot del buffer libre. Los consumidores toman los mensajes ejecutando *fetch*, demorando si es necesario hasta que haya un slot del buffer lleno.

El cap. 4 mostró como implementar un buffer limitado usando semáforos para sincronización. Aquí, usamos CCRs para implementar dos soluciones. En la primera, *deposit* y *fetch* tiene acceso exclusivo al buffer. En la segunda, *deposit* y *fetch* pueden ejecutar concurrentemente. En ambas soluciones, usamos CCRs directamente para obtener la sincronización requerida.

Buffer limitado con acceso exclusivo

Como en el cap. 4, representaremos el buffer por un arreglo *buf*[1:n] y dos variables índice, *front* y *rear*. Los mensajes que han sido depositados pero aún no buscados son almacenados en la cola circular comenzando en *buf*[front]. A menos que el buffer esté lleno, *buf*[rear] es el primer slot vacío pasando los mensajes depositados. Antes, sincronizábamos el acceso al buffer usando contadores que registraban el número de veces que fueron ejecutadas *deposit* y *fetch*. Esa aproximación es apropiada cuando se usan semáforos para implementar sincronización. Sin embargo, es más simple especificar la sincronización requerida usando una única variable, *count*, que registra el número de slots llenos. Los valores de todas estas variables están relacionadas por el predicado:

```
BUFFER: 1 ≤ front ≤ n  ∧  1 ≤ rear ≤ n  ∧  0 ≤ count ≤ n  ∧
            rear = (front + count - 1) mod n + 1  ∧
            count slots en la cola circular comenzando en buf[front]
                contienen los ítems más recientemente depositados en el orden
                en el cual fueron depositados
```

Este predicado debe ser invariante. La relación de orden en BUFFER podría ser especificada formalmente introduciendo algo como timestamps para registrar cuándo fue depositado cada ítem. Sin embargo, la especificación informal es más fácil de entender.

Inicialmente *count* es 0. Es incrementada en *deposit* y decrementada en *fetch*. Dados los límites sobre *count* especificados en BUFFER, *deposit* debe demorarse hasta que *count* sea menor que *n*, entonces las acciones de *deposit* son:

```
⟨ await  count < n → buf[rear] := data
                     rear := rear mod n + 1; count := count + 1 ⟩
```

Similarmente, *fetch* debe demorarse hasta que *count* sea positivo, entonces sus acciones son:

```
⟨ await  count > 0 → result := buf[front]
                     front := front mod n + 1; count := count - 1 ⟩
```

Nuevamente, es directo implementar estos fragmentos de código usando CCRs. Como antes, las variables compartidas se ponen en un recurso, BUFFER se convierte en el resource invariant, y los **await** son reemplazados por sentencias **region**. Esto lleva a la implementación:

```
resource buffer(buf[1:n[ : T; front := 1; rear := 1; count := 0)
{ BUFFER }
deposit: region buffer when count < n →
            buf[rear] := data; rear := rear mod n + 1; count := count + 1
         end
fetch: region buffer when count > 0 →
            result := buf[front]; front := front mod n + 1; count := cont - 1
       end
```

Claramente esta solución preserva invariante a BUFFER. También está libre de deadlock, asumiendo que el buffer tiene al menos un slot. Sin embargo, la solución impone más exclusión que la necesaria. En particular, tanto *deposit* como *fetch* tienen acceso exclusivo al buffer, aunque puedan ser ejecutados en forma segura en paralelo cuando hay slots vacíos y mensajes disponibles. En general, puede haber más exclusión que la necesaria cuando todas las variables compartidas están en el mismo recurso. Evitar la exclusión excesiva requiere remover alguna de las variables compartidas de los recursos o separarlos en recursos disjuntos. La próxima sección muestra cómo hacer esto.

Buffer limitado con acceso concurrente

En la implementación basada en semáforos de un buffer limitado, un productor y un consumidor pueden acceder concurrentemente a *buf*. Esa solución podría ser programada usando CCRs para simular semáforos. Sin embargo, el buffer en sí mismo no estaría dentro de un recurso. Esto viola el requerimiento impuesto por CCRs de que todas las variables compartidas pertenecen a recursos, y es ese requerimiento el que obvia la necesidad de probar que los procesos no interfieren.

Esta sección desarrolla una implementación basada en CCR en la cual todas las variables compartidas están dentro de recursos. La solución también ilustra el uso de sentencias **region** anidadas. La clave está en reespecificar el invariante del buffer en una manera distinta que no requiere una implementación con acceso exclusivo.

Independientemente de cómo es representado un buffer limitado, los requerimientos esenciales son que *deposit* no sobreescriba un mensaje no buscado, que *fetch* solo lea mensajes depositados y solo una vez, y que *fecht* lea mensajes en el orden en el cual fueron depositados. Si el buffer es representado por una cola circular, estos requerimientos se cumplen si los slots llenos son contiguos, *deposit* escribe en el próximo slot vacío (el cual se convierte en lleno), y *fetch* lee del primer slot lleno(el cual se convierte en vacío). En particular, representemos el buffer por:

```
var slot[1:n] : T, full[1:n] : bool := ( [n] false )
var front := 1, rear := 1
```

Aquí, *full* es un arreglo que registra para cada slot si está lleno o vacío. Ahora *deposit* y *fecht* deben mantener el invariante:

```
CONCBUF: 1 ≤ front ≤ n  ∧  1 ≤ rear ≤ n  ∧  los slots llenos en la cola
            circular comenzando en slot[front] contienen los mensajes
            más recientemente depositados en orden cronológico.
```

Como antes, la ejecución de *deposit* necesita demorarse si el buffer está lleno. Con la representación anterior, como especificaba BUFFER, el buffer estaba lleno si *count* era *n*. Aquí, el buffer está lleno si todos los slots están llenos; esto se simplifica a chequear si full[rear] es true. Así, tenemos la representación de *deposit* usando **await**:

```
⟨ await not full[rear] → slot[rear] := data; full[rear] := true
                         rear := rear mod n + 1 ⟩
```

Análogamente, la ejecución de *fetch* necesita demorarse si el buffer está vacío. Con la anterior representación, esto ocurre si *full*[front] es false, lo que da la representación de *fetch*:

⟨ **await** full[front] → result := slot[front]; full[front] := false
front := front **mod** n + 1 ⟩

Como es usual, podríamos convertir esta solución directamente en una que use CCRs poniendo todas las variables compartidas en un recurso e implementando *deposit* y *fetch* con sentencias **region**. La implementación resultante sería esencialmente equivalente a la de la sección previa. Sin embargo, podemos implementarlo de una manera distinta que permita el acceso concurrente a diferentes slots del buffer.

Para permitir más concurrencia en un programa, necesitamos buscar maneras de hacer las acciones atómicas finer-grained sin introducir interferencia. Primero consideremos la sentencia **await** en *deposit*. Solo las dos primeras asignaciones necesitan ser guardadas. La última sentencia, que modifica *rear*, no necesita serlo pues incrementar *rear* módulo *n* asegura que *rear* se mantiene entre 1 y *n*, como especificamos en CONCBUF. Así la sentencia **await** en *deposit* puede ser separada en dos acciones atómicas (una para manipular *slot* y *full*, la otra para incrementar *rear*):

(5.5) ⟨ **await not** full[rear] → slot[rear] := data; full[rear] := true⟩
⟨ rear := rear **mod** n + 1 ⟩

Las mismas observaciones se aplican a *fetch*. No necesitamos guardar la asignación a *front*, de modo que puede ser ubicada en una acción atómica separada:

(5.5) ⟨ **await** full[front] → result := slot[front]; full[front] := false⟩
⟨ front := front **mod** n + 1 ⟩

Cuando un proceso productor está entre las dos acciones atómicas en *deposit*, un slot más fue llenado, pero *rear* aún no fue actualizado. En este punto, *full*[rear] es true, y entonces un segundo productor se demorará hasta que *rear* sea incrementado. Esto asegura que CONCBUF es invariante pues asegura que todos los slots llenos son contiguos y que los mensajes son depositados en orden cronológico.

En (5.5), el primer **await** en *deposit* referencia solo un slot del buffer, y el último referencia solo una de las variables punteros; las sentencias **await** en *fetch* (5.6) son similares. Como vimos antes, la manera de incrementar la concurrencia con CCRs es poner variables compartidas en distintos recursos. Aquí podemos emplear varios recursos disjuntos: uno para cada slot, uno para *front*, y uno para *rear*. Entonces parecería que podemos implementar (5.5) y (5.6) usando cuatro sentencias **region** distintas, una para cada acción atómica. Sin embargo, las reglas para usar CCRs requieren que todas las variables en un recurso sean accedidas solo dentro de sentencias **region** que nombran el recurso. Ambas acciones en *deposit* en (5.5) requieren acceso a *rear*; análogamente, ambas acciones en *fetch* en (5.6) requieren acceso a *front*. Para programar esto usando CCRs, anidamos una sentencia **region** para acceder a un slot del buffer dentro de una **region** que accede uno de los punteros.

Basado en estas consideraciones, tenemos el siguiente programa basado en CCRs (La solución es correcta, independiente del número de productores o consumidores. Sin embargo, a lo sumo un productor y consumidor a la vez pueden estar accediendo al buffer. Si no es imperativo que los mensajes sean depositados y buscados en orden estrictamente cronológico, la solución puede convertirse en una en que todos los slots del buffer pueden ser accedidos concurrentemente):

```
resource buffer[1:n] (slot : T; full : bool := false)
resource f (front : int := 1 )
resource r (rear : int := 1 )
{ CONCBUF }
deposit: region r →
            region buffer[rear] when not full →
              slot := data; full := true
            end
```

```
            rear := rear mod n + 1
        end
fetch: region f →
            region buffer[front] when full →
              result := slot; full := false
            end
            front := front mod n + 1
        end
```

Aunque esta implementación permite a los productores y consumidores acceder al buffer concurrentemente, generalmente será menos eficiente que la de la sección anterior. Esto es porque contiene dos sentencias **region** tanto en *deposit* como *fetch*, y las sentencias *region* son relativamente caras para implementar. Así, esta última solución es preferible a la anterior solo si el tiempo gastado en acceder al buffer es al menos tan grande como el overhead de sincronización impuesto por las sentencias **region**. Este será el caso solo si toma un largo tiempo llenar o vaciar un slot del buffer, o porque los slots son grandes o porque están en almacenamiento externo (y entonces las asignaciones al buffer son realmente transferencia de archivos).

SCHEDULING Y ALOCACION DE RECURSOS

En el cap. 4, describimos la naturaleza general de los problemas de alocación de recursos y mostramos cómo se podrían usar semáforos para resolverlos. En particular, derivamos una solución al problema de alocación SJN. Nuevamente, podríamos reprogramar esa solución usando CCRs para simular semáforos. Como con otros problemas considerados en este capítulo, hay una solución más directa usando CCRs. La idea básica es la misma que usamos en la solución de preferencia de escritores: tener un proceso esperando por una condición distinta.

Recordemos que, en el problema de alocación SJN, varios procesos compiten por el acceso a un recurso compartido. Los procesos requieren el uso del recurso, lo usan una vez que su pedido fue atendido, y eventualmente lo retornan. Si más de un proceso es demorado esperando a que se satisfaga su pedido, cuando el recurso es liberado es otorgado al proceso demorado que lo usará el menor tiempo. Como antes, sea *free* una variable booleana que indica si el recurso está libre o en uso, y sea *P* un conjunto de pares (*time*, *id*) ordenados por los valores de *time*. Luego, como antes, el siguiente predicado expresa los requerimientos de una política de alocación SJN:

SJN: P en un conjunto ordenado $\wedge$ (P = $\varnothing$)

Ignorando la política de alocación por ahora, la solución coarse-grained es:

```
var free := true
request(time,id): ⟨ await free → free := false ⟩
release( ): ⟨ free := true ⟩
```

Podríamos usar CCRs para implementar esta solución directamente, pero entonces no ocurriría el scheduling: Cuando el recurso es liberado, sería adquirido por un demorado arbitrario. Con semáforos, usábamos el método de passing the baton para implementar los fragmentos de programa de modo que se sigue la política SJN. Esa aproximación no puede ser usada con CCRs, a menos que solo usemos CCRs para simular semáforos. Hay dos razones. Primero, a un proceso no se le permite demorarse en el medio de una sentencia **region**; solo puede hacerlo antes de entrar. Usar una segunda **region** anidada no ayuda pues un proceso demorado en la sentencia anidada retendría el control exclusivo del recurso nombrado en la sentencia externa. Segundo, cuando un proceso sale de una sentencia **region**, libera el control del recurso a cualquier otro proceso que esté demorado en una sentencia **region** cuya guarda es true. Las CCRs no proveen directamente ningún análogo de SBS (que provee la base de passing the baton). Pero, la idea subyacente en el método de passing the baton sugiere cómo resolver el problema de alocación SJN usando CCRs. La clave es asignar una condición de demora única a cada proceso.

Un proceso que hace un pedido primero necesita chequear si el recurso está libre. Si lo está, el proceso puede tomarlo; si no, el proceso necesita salvar sus argumentos (*time*, *id*) en el conjunto *P* y luego demorarse. Dado que *P* es compartido, debe pertenecer a un recurso y entonces solo puede ser accedido usando sentencias **region**. Así, necesitamos usar una sentencia **region** para almacenar los argumentos en *P*, y una segunda **region** para demorarse. Dado que el estado del recurso podría cambiar entre la ejecución de las dos sentencias **region**, la salida de la primera (si el proceso puede tomar el recurso o necesita demorar) debe ser registrada. Podemos programar esto usando una variable *next* que es asignada con el valor de *id* del próximo proceso al que se alocará el recurso. En particular, si un proceso encuentra que el recurso está *free*, setea *next* a su *id*; en otro caso, almacena (*time*,*id*) en el lugar apropiado de *P*; luego el proceso se demora en la segunda **region** hasta que *next* es igual a su *id*.

Consideremos *release*. El requerimiento clave es “pasar el baton” a un pedido demorado o liberar el recurso si no hay pedidos demorados. Dado que las CCRs usan guardas booleanas en las sentencias **region**, tenemos que “pasar el baton” usando una expresión booleana más que un semáforo. Nuevamente, podemos usar una variable *next*. Cuando el recurso es liberado y *P* no está vacío, se remueve el primer pedido pendiente, y *next* es seteado al *id* de ese pedido. En otro caso, *next* se setea a algún valor que no es un identificador de proceso, y *free* se setea en true. Asumiremos que los identificadores de proceso son positivos, y setearemos *next* a 0 cuando el recurso está libre. La solución con CCRs usando esta aproximación es la siguiente:

```
resource sjn (free : bool := true; next := 0; P : set of (int, int) := ∅)
{ SJN: P en un conjunto ordenado ∧ (P = ∅) }
request(time,id): region sjn →
                      if free → free := false; next := id
                      □ not free → insert(time,id) en P
                      fi
                  end
                  region sjn when next = id → skip end
release( ): region sjn →
              if P ≠ ∅ → remover el primer par (time,id) de P; next := id
              □ P = ∅ → free := true; next := 0
              fi
            end
```

La estructura es similar a la solución con semáforos. Pero esta solución usa *next* y condiciones booleanas en lugar de semáforos para indicar cuál proceso es el próximo en usar el recurso. Otra diferencia es la necesidad de dos **region** en *request*. Esto es porque un proceso no puede demorarse dentro de una **region**. Por otro lado, el uso de sentencias **region** asegura que las variables compartidas son accedidas con exclusión mutua. Con semáforos, se debe tener cuidado de asegurar que el se mantiene la propiedad de SBS.

Podemos usar este patrón para cualquier problema de alocación. Podemos implementar distintas políticas imponiendo distintos órdenes sobre la cola de request y usando múltiples colas de request si hay múltiples recursos siendo alocados a la vez. La clave es usar valores únicos como base para las condiciones de demora.

Monitores

Con semáforos y CCRs, las variables compartidas son globales a todos los procesos. Las sentencias que acceden variables compartidas podrían ser agrupadas en procedures, pero tales sentencias pueden estar dispersadas en un programa. Entonces, para entender cómo se usan las variables compartidas, se debe examinar un programa entera. Además, si un nuevo proceso se agrega al programa, el programador debe verificar que el proceso usa las variables compartidas correctamente.

Las CCRs son más estructuradas que los semáforos pues las variables relacionadas están declaradas juntas en un recurso con un invariante asociado y las variables del recurso pueden ser accedidas solo por medio de sentencias **region**. Sin embargo, las **region** son mucho más costosas de implementar que las operaciones semáforo.

Los *monitores* son módulos de programa que proveen más estructura que las CCRs y pueden ser implementadas tan eficientemente como los semáforos. Antes que nada, los monitores son un mecanismo de abstracción de datos: encapsulan las representaciones de recursos abstractos y proveen un conjunto de operaciones que son los *únicos* medios por los cuales la representación es manipulada. En particular, un monitor contiene variables que almacenan el estado del recurso y procedimientos que implementan operaciones sobre el recurso. Un proceso puede acceder las variables en un monitor solo llamando a uno de los procedures del monitor. La exclusión mutua es provista asegurando que la ejecución de procedures en el mismo monitor no se overlapea. Esto es similar a la exclusión mutua implícita provista por las sentencias **region**. Sin embargo, en los monitores la sincronización por condición es provista por un mecanismo de bajo nivel llamado *variables condición* (condition variables). Como veremos, estas variables son usadas para señalizar en forma bastante similar a los semáforos.

Cuando los monitores son usados para sincronización, un programa concurrente contiene dos clases de módulos: procesos activos y monitores pasivos. Asumiendo que todas las variables compartidas están dentro de monitores, dos procesos pueden interactuar sólo llamando procedures en el mismo monitor. La modularización resultante tiene dos beneficios importantes. Primero, un proceso que llama a un procedure de monitor puede ignorar cómo es implementado el procedure; todo lo que importa son los efectos visibles de llamar al procedure. Segundo, el programador de un monitor puede ignorar cómo o dónde se usan los procedures del monitor. Una vez que un monitor es implementado correctamente, se mantiene correcto, independientemente del número de procesos que lo usan. Además, el programador de un monitor es libre de cambiar la manera en la cual el monitor es implementado, mientras los procedures visibles y sus efectos no cambien. Estos beneficios hacen posible diseñar cada proceso y el monitor en forma relativamente independiente. Esto hace al programa concurrente más fácil de desarrollar y entender.

Este capítulo describe monitores, discute su semántica e implementación, e ilustra su uso con varios ejemplos. Se introducen y resuelven varios problemas nuevos, incluyendo dos básicos: el sleeping barber y el scheduling de disk head. A causa de su utilidad y eficiencia, los monitores fueron empleados en varios lenguajes de programación concurrentes, incluyendo Pascal Concurrente, Modula, Mesa, Pascal Plus, Euclid Concurrente, Touring Plus.

NOTACION

Un monitor pone una pared alrededor de las variables y procedures que implementan un recurso compartido. Una declaración de monitor tiene la forma:

(6.1) **monitor** *Mname*
declaraciones de variables permanentes; código de inicialización
procedure op_1 (par. formales$_1$) cuerpo de op_1 **end**
.....
procedure op_n (par. formales$_n$) cuerpo de op_n **end**
end

Las variables permanentes representan el estado del recurso. Son llamadas así pues existen y mantiene sus valores mientras existe el monitor. Los procedures implementan las operaciones sobre el recurso. Como es usual, estos procedures pueden tener parámetros formales y variables locales.

Un monitor tiene tres propiedades que son consecuencia de ser un TAD. Primero, solo los nombres de los procedures son visibles fuera del monitor (proveen las únicas puertas a través de la pared del monitor). Así, para alterar el estado del recurso representado por las variables permanentes, un proceso debe llamar a uno de los procedures del monitor. Sintácticamente, los llamados al monitor tienen la forma:

call *Mname*.*op*$_i$ (argumentos)

donde *op*$_i$ es uno de los procedures de *Mname.* La segunda propiedad es que los procedures dentro de un monitor pueden acceder solo las variables permanentes y sus parámetros y variables locales. No pueden acceder variables declaradas fuera del monitor. Tercero, las variables permanentes son inicializadas antes de que se ejecute cualquier cuerpo de procedure.

Uno de los atributos de un monitor (o cualquier TAD) es que puede ser desarrollado en relativo aislamiento. Sin embargo, como consecuencia, el programador de un monitor no puede conocer *a priori* el orden en el cual serán llamados los procedures del monitor. Como es usual, siempre que el orden de ejecución es indeterminado, necesitamos definir un invariante. Aquí el invariante es un *monitor invariant* que especifica los estados “razonables” de las variables permanentes cuando ningún proceso las está accediendo. El código de inicialización en un monitor debe establecer el invariante. Cada procedure debe mantenerlo.

Lo que distingue a un monitor de un mecanismo de abstracción de datos en un lenguaje de programación secuencial es que un monitor es compartido por procesos que ejecutan concurrentemente. Así, los procesos que ejecutan en monitores pueden requerir exclusión mutua (para evitar interferencia) y pueden requerir sincronización por condición (para demorarse hasta que el estado del monitor sea conductivo a ejecución continuada. Ahora pondremos atención en cómo los procesos sincronizan dentro de los monitores.

Sincronización en Monitores

La sincronización en monitores podría ser provista de varias maneras. Aquí se describe un método empleado en los lenguajes Mesa y Turing Plus y en el SO Unix.

La sincronización es más fácil de entender y de programar si la exclusión mutua y la sincronización por condición son provistas de distintas maneras. Es mejor si la exclusión mutua es provista implícitamente pues esto evita interferencia. En cambio, la sincronización por condición debe ser explícitamente programada pues distintos programas requieren distintas condiciones de sincronización. Aunque es más fácil sincronizar por medio de condiciones booleanas como en las sentencias **region**, los mecanismos de más bajo nivel pueden ser implementados mucho más eficientemente. También proveen al programador control más fino sobre el orden de ejecución, lo cual ayuda a resolver problemas de alocación y scheduling.

Basado en estas consideraciones, la exclusión mutua en monitores es provista implícitamente y la sincronización por condición es programada explícitamente usando mecanismos llamados variables condición. En particular, a lo sumo un proceso a la vez puede estar ejecutando dentro de cualquier procedure de un monitor. Sin embargo, dos procesos pueden ejecutar concurrentemente fuera de los monitores o en distintos monitores. Dado que la ejecución dentro de un monitor es mutuamente exclusiva, los procesos no pueden interferir cuando acceden a las variables permanentes.

Una *variable condición* se usa para demorar un proceso que no puede seguir ejecutando en forma segura hasta que el estado del monitor satisfaga alguna condición booleana. También se usa para despertar a un proceso demorado cuando la condición se convierte en true. La declaración de una variable condición tiene la forma:

var c : **cond**

Un arreglo de variables condición se declara de la forma usual agregando información de rango al nombre de la variable. El valor de *c* es una cola de procesos demorados, pero este valor no es visible directamente al programador. La condición de demora booleana está implícitamente asociada con la variable condición por el programador; seguiremos la convención de especificarlo en un comentario en la declaración de la variable. Dado que las variables condición se usan para sincronizar el acceso a variables permanentes de un monitor, deben ser declaradas y usadas solo dentro de monitores.

Para demorarse sobre una variable condición *c* un proceso ejecuta:

wait(c)

La ejecución de **wait** causa que el proceso se demore al final de la cola de *c*. Para que algún otro proceso pueda eventualmente entrar al monitor para despertar al proceso demorado, la ejecución de **wait** también causa que el proceso deje el acceso exclusivo al monitor. (Si un proceso ejecutando en un monitor llama a un procedure de un segundo monitor y luego espera en ese procedure, libera la exclusión solo en el segundo monitor. El proceso retiene el control exclusivo del primer monitor).

Los procesos demorados sobre variables condición son despertados por medio de sentencias **signal**. Si la cola de demorados de *c* no está vacía, la ejecución de

signal(c)

despierta al proceso que está al frente de la cola y lo remueve de la cola. Ese proceso ejecuta en algún tiempo futuro cuando pueda readquirir el acceso exclusivo al monitor. Si la cola de *c* está vacía, la ejecución de **signal** no tiene efecto; es decir, es equivalente a **skip**. Independiente de si un proceso demorado es despertado, el proceso que ejecuta **signal** retiene el control exclusivo del monitor; así puede continuar ejecutando. Por esta razón, se dice que **signal** tiene semántica *signal-and-continue* (existen semánticas de señalización alternativas).

Las operaciones **wait** y **signal** sobre variables condición son similares a las **P** y **V** sobre semáforos. Sin embargo, hay tres diferencias importantes. Primero, **signal** no tiene efecto si ningún proceso está demorado sobre la variable condición; el hecho de que fue ejecutada no se recuerda. Segundo, **wait** siempre demora un proceso hasta que se ejecute un **signal** posterior. Tercero, el proceso que ejecuta **signal** siempre ejecuta (al menos en el monitor) antes que un proceso despertado como resultado del **signal**. Estas diferencias causan que la sincronización por condición sea programada en forma diferente con variables condición que con semáforos.

Podemos implementar una solución al problema del buffer limitado usando monitores. (En particular, el invariante del monitor es el mismo predicado BUFFER que en el cap. 5 se usaba como invariante del recurso):

```
monitor Bounded_Buffer
 var buf[1:n] : T
 var front := 1, rear := 1, count := 0
 var not_full : cond        # signal cuando count < n
 var not_empty : cond       # signal cuando count > 0
 { BUFFER: 1 ≤ front ≤ n  ∧  1 ≤ rear ≤ n  ∧  0 ≤ count ≤ n  ∧
           rear = (front + count - 1) mod n + 1  ∧
           count slots en la cola circular comenzando en buf[front]
              contienen los ítems más recientemente depositados en el orden
              en el cual fueron depositados }
 procedure deposit (data : T )
  do count = n → wait(not_full) od
  buf[rear] := data; rear := ( rear mod n ) + 1; count := count + 1
  signal(not_empty)
 end
 procedure fetch (var result : T )
```

```
    do count = 0 → wait(not_empty) od
    result := buf[front]; front := ( front mod n ) + 1; count := count - 1
    signal(not_full)
  end
end
```

Aquí, ambos **wait** están dentro de loops. Esta siempre es una manera de asegurar que la condición deseada es true antes de que se accedan las variables permanentes. Aquí es necesario si hay múltiples productores y consumidores. Cuando un proceso ejecuta **signal**, solo avisa que la condición señalada ahora es true. Dado que el señalador y posiblemente otros procesos pueden ejecutar en el monitor antes de que un proceso despertado por **signal**, la condición puede ya no ser válida cuando el proceso despertado reanuda la ejecución. Por ejemplo, un productor podría ser demorado esperando un slot vacío, luego un consumidor podría buscar un mensaje y despertar al productor dormido. Sin embargo, antes de que el productor tome el turno para ejecutar, otro productor podría entrar a *deposit* y llenar el slot vacío. Una situación análogo podría ocurrir con los consumidores. Así, en general es necesario rechequear la condición de demora.

Las sentencias **signal** en *deposit* y *fetch* son ejecutadas incondicionalmente pues en ambos casos la condición señalada es true en el momento del **signal**. De hecho, si los **wait** son encerrados en loops que rechequean la condición esperado, los **signal** pueden ser ejecutados *en cualquier momento* pues solo dan un aviso a los procesos demorados. Sin embargo, un programa ejecutará más eficientemente si **signal** es ejecutado solo si es cierto que algún proceso demorado podría continuar. En resumen, usualmente es seguro ejecutar **signal** con más frecuencia que la necesaria, y debe ser ejecutada con la suficiente frecuencia para evitar deadlock y demora innecesaria.

Operaciones adicionales sobre Variables Condición

Algunas operaciones adicionales sobre variables condición son útiles. Todas tienen semántica simple y pueden ser implementadas eficientemente pues solo proveen operaciones adicionales sobre la cola asociada a la variable condición.

Para determinar si al menos un proceso está demorado sobre la variable condición *c*, un proceso en un monitor puede invocar la función booleana:

empty(c)

Esta función es true si la cola está vacía; en otro caso da falso.

Con **wait** y **signal** como definimos, los procesos demorados son despertados en el orden en que fueron demorados; es decir, la cola de demora es FIFO. La sentencia **wait** *con prioridad* permite al programador tener más control sobre el orden en el cual los procesos demorados son encolados, y por lo tanto despertados. Esta sentencia tiene la forma:

wait(c,rank)

Aquí, *c* es una variable condición y *rank* es una expresión entera. Los procesos demorados en *c* son despertados en orden ascendente de *rank*; el proceso demorado hace más tiempo es despertado en caso de empate. Para evitar la potencial confusión resultante de usar **wait** regulares y con prioridad sobre la misma variable condición, siempre usaremos solo una clase.

Usando **empty** y **wait** con prioridades es directo implementar un monitor que provea alocación SJN para un recurso de una sola unidad:

```
monitor Shortest_Job_Next
  var free := true
  var turn : cond       # signal cuando el recurso está disponible
  { SJN: turn está ordenada por time  ∧  free ⇒ turn está vacío }
  procedure request (time : int )
    if free → free := false □ not free → wait(turn,time) fi
```

```
  end
  procedure release ( )
    if empty(turn) → free := true □ not empty(turn) → signal(turn) fi
  end
end
```

El monitor tiene dos operaciones: *request* y *release*. Cuando un proceso llama a *request*, se demora hasta que el recurso esté libre o se le aloca a él. Luego de adquirir y usar el recurso, un proceso llama a *release*. El recurso luego es alocado al proceso que espera y tiene el menor *time*; si no hay pedidos pendientes, el recurso es liberado.

En este ejemplo, se usa **wait** con prioridad para ordenar los procesos demorados por la cantidad de tiempo que usarán el recurso; **empty** se usa para determinar si hay procesos demorados. Cuando el recurso es liberado, si hay procesos demorados, el que tiene mínimo *rank* es despertado; en otro caso, el recurso es marcado como libre. En este caso el **wait** no es puesto en un loop pues la decisión de cuándo puede continuar un proceso demorado es hecha por el proceso que libera el recurso.

Con **wait** con prioridad, algunas veces es útil determinar el ranking del proceso al frente de la cola, es decir, el valor del mínimo ranking de demora. Asumiendo que la cola de demora asociada con *c* no está vacía y todas los **wait** que referencian *c* son con prioridad,

minrank(c)

es una función entera que retorna el mínimo ranking de demora. Si la cola está vacía, **minrank** retorna algún valor entero arbitrario.

El **signal** *broadcast* es la última operación sobre variables condición. Se usa si más de un proceso demorado podría seguir o si el señalador no sabe cuáles procesos demorados podrían seguir (pues ellos mismos necesitan rechequear sus condiciones de demora). Esta operación tiene la forma:

signal_all(c)

La ejecución de **signal_all** despierta a todos los procesos demorados sobre *c*. En particular, su efecto es el mismo que ejecutar:

do not empty(c) → **signal**(c) **od**

Cada proceso despertado reanuda la ejecución en el monitor en algún momento futuro, sujeto a la restricción de exclusión mutua. Como **signal**, **signal_all** no tiene efecto si ningún proceso está demorado sobre *c*. También, el proceso que señaliza continúa ejecutando en el monitor.

Siempre que los **wait** que nombran una variable condición estén en loops que rechequean la condición, **signal_all** puede usarse en lugar de **signal**. Dado que los procesos despertados ejecutan con exclusión mutua y rechequean sus condiciones de demora, los que encuentran que la condición ya no es true simplemente vuelve a dormirse. Por ejemplo, **signal_all** podría ser usada en lugar de **signal** en el monitor del buffer. Sin embargo, en este caso es más eficiente usar **signal** pues a lo sumo un proceso despertado podría seguir; los otros deberían volver a dormir.

Semántica formal y prueba de programas

Un programa concurrente que emplea monitores contendrá procesos y monitores. Para desarrollar una prueba formal de tal programa, necesitamos desarrollar pruebas para cada proceso y cada monitor y luego juntar las pruebas. Las pruebas de los procesos se desarrollan como antes. Las pruebas de monitor son desarrolladas empleando axiomas y reglas de inferencia para los mecanismos del monitor. Las pruebas separadas son juntadas usando una regla de inferencia para el llamado a procedure.

Implementación de semáforos usando monitores

Consideremos el problema de usar un monitor para implementar un semáforo general. Sea *s* un entero que registra el valor del semáforo y sean *P* y *V* procedures que implementan las operaciones **P** y **V** sobre el semáforo. Como es usual, *P* y *V* deben preservar el invariante del semáforo $s \geq 0$, el cual en este caso sirve como invariante del monitor; por lo tanto, *P* debe demorarse hasta que *s* sea positivo. Una implementación de tal monitor es la siguiente:

(6.6)
```
monitor Semaphore        # Invariante SEM: s ≥ 0
  var s := 0, pos : cond    # pos es señalizada cuando s > 0
  procedure P( ) do s = 0 → wait(pos) od; s := s - 1 end
  procedure V( ) s := s + 1; signal(pos) end
end
```

El loop **do** en *P* asegura que *s* es positiva antes de ser decrementada. El **signal** en *V* despierta un proceso demorado, si hay uno.

Lectores y Escritores

Esta sección presenta una solución al problema de lectores/escritores usando monitores. Recordemos que los lectores consultan una BD y los procesos escritores la examinan y alteran. Los lectores pueden acceder la BD concurrentemente, pero los escritores necesitan acceso exclusivo.

Aunque la BD es compartida, no podemos encapsularla por un monitor pues entonces los lectores no podrían acceder la BD concurrentemente. En lugar de esto, usamos un monitor solo para arbitrar el acceso a la BD. La BD en sí misma es global a los lectores y escritores (por ej, está almacenada en un archivo externo). Como veremos, la misma estructura básica se usa con frecuencia en programas basados en monitores.

Para este problema, el arbitraje del monitor concede permiso para acceder la BD. Para hacerlo, se requiere que los procesos informen cuando quieren acceder y cuando terminaron el acceso. Dado que hay dos clases de procesos y dos acciones por proceso, el monitor tiene 4 procedures: *request_read*, *release_read*, *request_write* y *release_write*. Estos procedures se usan de las maneras obvias. Por ejemplo, un lector llama a *request_read* antes de leer la BD y llama *release_read* después de leer la BD.

Para sincronizar el acceso a la BD, necesitamos registrar cuántos procesos están leyendo y cuántos escribiendo. Así, como antes, sea *nr* el número de lectores y *nw* el de escritores. Estas son las variables permanentes del monitor. Para una sincronización adecuada, estas variables deben satisfacer el invariante del monitor:

$$RW: (nr = 0 \vee nw = 0) \wedge nw \leq 1$$

Inicialmente, *nr* y *nw* son 0. Cada variable es incrementada en el procedure request adecuado y decrementada en el release apropiado.

El siguiente monitor cumple esta especificación:

```
monitor RW_Controller              # Invariante RW
  var nr := 0, nw := 0
  var oktoread : cond              # signal cuando nw = 0
  var oktowrite : cond             # signal cuando nr = 0 ∧ nw = 0
  procedure request_read( )
    do nw > 0 → wait(oktoread) od
    nr := nr + 1
  end
  procedure release_read( )
    nr := nr - 1
    if nr = 0 → signal(oktowrite) fi
```

```
  end
  procedure request_write( )
    do nr > 0 ∨ nw > 0 → wait(oktowrite) od
    nw := nw + 1
  end
  procedure release_write( )
    nw := nw - 1
    signal(oktowrite)
    signal_all(oktoread)
  end
end
```

Las variables condición y las sentencias **wait** y **signal** son usadas para asegurar que RW es invariante. Al comienzo de *request_read*, un proceso lector necesita demorarse hasta que *nw* no sea 0. Sea *oktoread* la variable condición sobre la cual se demoran los lectores. Análogamente, los escritores necesitan demorarse al comienzo de *request_write* hasta que *nr* y *nw* sean 0. Sea *oktowrite* la variable condición sobre la cual se demoran. Cuando *nr* se convierte en 0 en *release_read*, un escritor demorado puede seguir, si hay uno. Despertamos un escritor demorado señalizando *oktowrite*. Cuando *nw* se vuelve 0 en *release_write*, un escritor demorado o todos los lectores demorados pueden seguir. Despertamos un escritor demorado señalizando *oktowrite*; despertamos todos los lectores con un signal broadcast de *oktoread*.

En el ejemplo, los escritores son señalizados en *release_read* solo si *nr* es 0. Dado que los escritores rechequean su condición de demora, la solución aún sería correcta si los escritores fueran siempre señalizados. Sin embargo, la solución sería menos eficiente pues un escritor señalizado debería volver a dormirse si *nr* no era 0. Por otra parte, al final de *release_write*, se sabe que tanto *nr* como *nw* son 0. Entonces cualquier proceso demorado podría continuar. La solución no arbitra entre lectores y escritores. Despierta a todos los procesos dormidos y deja a la política de scheduling subyacente determinar cuál ejecuta primero y toma el acceso a la BD.

TECNICAS DE SINCRONIZACION

En esta sección se desarrollan soluciones basadas en monitores a tres problemas de sincronización. Con esto ilustraremos varias técnicas de programación distintas. El método de derivación emplea los mismos pasos básicos usados en mecanismos de sincronización previos: definir el problema y la estructura de solución, anotar la solución, guardar las asignaciones, e implementar la solución resultante usando monitores. Sin embargo, cada paso se realiza de manera un poco diferente debido a las características únicas de los monitores.

Dado que los monitores son un mecanismo de abstracción, el punto de partida al diseñar un monitor es definir el recurso abstracto que implementa. El mejor lugar para comenzar es con los procedures del monitor. En particular, primero definimos los nombres de los procedures, sus parámetros, las pre y postcondiciones de sus cuerpos, y cualquier suposición acerca del orden en que los procedures serán llamados por los procesos. Luego, especificamos las variables permanentes del monitor y el invariante asociado. Finalmente, hacemos un outline de los cuerpos de los procedures y agregamos código de inicialización para que el invariante sea true al inicio.

Luego de hacer un outline de los cuerpos de los procedures, necesitamos agregar sincronización por condición cuando es necesario asegurar el invariante del monitor o asegurar la postcondición deseada de un procedure. Cuando desarrollamos una solución coarse-grained, usaremos una variante de la sentencia **await** para especificar sincronización por condición. En particular, usaremos **await** *B*, donde *B* es una condición booleana. La ejecución de esta forma de **await** demora al proceso ejecutante hasta que *B* sea true. Mientras está demorado, el proceso libera el control exclusivo del monitor; por lo tanto el invariante del monitor debe ser true cuando **await** *B* es ejecutado. Cuando el proceso continúa, el invariante del monitor será nuevamente true, como será *B*.

Para completar la implementación de un monitor, necesitamos usar variables condición para implementar las sentencias **await**. En general, usaremos una variable condición por cada condición diferente que aparezca en las sentencias **await**. Sea c_B la variable condición asociada con **await** *B*. Entonces, implementaremos generalmente **await** *B* con el siguiente loop:

do not B → **wait**(c_B) **od**

Sin embargo, como veremos, algunas veces podemos representar condiciones relacionadas con la misma variable condición, y a veces podemos usar **wait** con prioridad para ordenar procesos demorados. También, a veces es útil implementar **await** usando una sentencia **if** en lugar de un loop **do**.

Para despertar procesos demorados sobre variables condición, también necesitamos agregar sentencias **signal** a los cuerpos de los procedures. En particular, los agregamos después de asignaciones a variables permanentes siempre que alguna condición de demora podría ser verdadera y algún proceso podría estar esperando por esa condición. Una falla al ejecutar un número adecuado de **signals** podría resultar en bloqueo permanente de algunos procesos. Como notamos antes, la ejecución de **signals** superfluos no introduce errores cuando los **wait** están embebidos en loops. Sin embargo, un programa ejecutará más eficientemente si los **signal** son ejecutados solo cuando es posible que un proceso demorado pueda seguir.

Para resumir, los aspectos específicos del monitor del método de derivación general son el foco inicial sobre la abstracción provista por un monitor, el uso de **await** *B* para demorar la ejecución cuando es necesario, y el uso de variables condición para implementar las sentencias **await**. Aunque el programador experimentado agregará variables condición directamente a los cuerpos de los procedures, primero guardaremos las asignaciones usando **await** y luego usamos variables condición para implementar **await**. De esta manera, enfocamos primero las condiciones booleanas requeridas para una correcta sincronización y luego consideramos cómo implementar la sincronización eficientemente usando variables condición.

Un semáforo fair: Passing the Condition

A veces es necesario asegurar que un proceso despertado por un **signal** toma precedencia sobre otros procesos que llaman a un procedure del monitor antes de que el proceso despertado tenga chance de ejecutar. Esto se hace pasando una condición directamente a un proceso despertado en lugar de hacerla globalmente visible. Ilustramos la técnica derivando un monitor que implementa un semáforo fair: un semáforo en el cual los procesos son capaces de completar las operaciones **P** en orden FCFS.

Recordemos la implementación del monitor *Semaphore* ya vista. Allí, puede parecer que los procesos pueden completar *P* en el orden en el cual llamaron a *P*. Las colas de las variables condición son FIFO en ausencia de sentencias **wait** con prioridad. En consecuencia, los procesos demorados son despertados en el orden en que fueron demorados. Sin embargo, dado que **signal** es no preemptivo, antes de que un proceso despertado tome chance de ejecutar, algún otro proceso podría llamar a *P*, encontrar que s > 0, y completar *P* antes que el proceso despertado. En resumen, los llamados desde afuera del monitor pueden “robar” condiciones señalizadas.

Para evitar que las condiciones sean robadas, el proceso de señalización necesita pasar la condición directamente al proceso despertado más que hacerla globalmente visible. Podemos implementar esto como sigue. Primero, cambiamos el cuerpo de *V* a una sentencia alternativa. Si algún proceso está demorado sobre *pos* cuando *V* es llamado, despertamos uno pero *no* incrementamos *s*; en otro caso incrementamos *s*. Segundo, reemplazar el loop de demora en *P* por una sentencia alternativa que chequea la condición *s* > 0. Si *s* es positivo, el proceso decrementa *s* y sigue; en otro caso, el proceso se demora sobre *pos*. Cuando un proceso demorado sobre *pos* es despertado, solo retorna de *P*; no decrementa *s* en este caso para compensar el hecho de que *s* no fue incrementado antes de ejecutar **signal** en la operación *V*. Incorporando estos cambios tenemos el siguiente monitor:

```
monitor Semaphore        # Invariante SEM: s ≥ 0
  var s := 0
  var pos : cond     #  señalizada en V cuando pos no está vacía
  procedure P( )
    if s > 0 → s := s - 1
    □ s = 0 → wait(pos)
    fi
```

```
  end
  procedure V( )
    if empty(pos) → s := s + 1
    □ not empty(pos) → signal(pos)
    fi
  end
end
```

Aquí la condición asociada con *pos* cambió para reflejar la implementación diferente. Además, *s* ya no es la diferencia entre el número de operaciones *V* y *P* completadas. Ahora el valor de *s* es nV-nP-pending, donde nV es el número de operaciones *V* completadas, nP es el número de operaciones *P* completadas, y *pending* es el número de procesos que han sido despertados por la ejecución de **signal** pero aún no terminaron de ejecutar *P*. (Esencialmente, *pending* es una variable auxiliar implícita que es incrementada antes de **signal** y decrementada después de **wait**).

Esta técnica de pasar una condición directamente a un proceso que espera puede ser aplicada a muchos otros problemas. Sin embargo esta técnica debe ser usada con cuidado, pues las sentencias **wait** no están en loops que rechequean la condición. En lugar de esto, la tarea de establecer la condición y asegurar que se mantendrá verdadera está en el señalador. Además, las sentencias **signal** no pueden ya ser reemplazadas por **signal_all**.

El problema del Sleeping Barber: Rendezvous

En esta sección, consideramos otro problema clásico de sincronización: el sleeping barber. Como en el problema de los filósofos, tiene una definición “colorida”. Más aún, es representativo de problemas prácticos, tales como el scheduler de disco que describiremos. En particular, el problema ilustra la importante relación cliente/servidor que con frecuencia existe entre procesos. También, contiene un tipo importante de sincronización llamada rendezvous. Finalmente, el problema es una ilustración excelente de la utilidad y necesidad de una aproximación sistemática para resolver problemas de sincronización.

(6.16) **Problema del sleeping barber**. Una ciudad tiene una pequeña peluquería con dos puertas y unas pocas sillas. Los cliente entran por una puerta y salen por la otra. Dado que el negocio es chico, a lo sumo un cliente o el peluquero se pueden mover en él a la vez. El peluquero pasa su vida atendiendo clientes, uno por vez. Cuando no hay ninguno, el peluquero duerme en su silla. Cuando llega un cliente y encuentra que el peluquero está durmiendo, el cliente lo despierta, se sienta en la silla del peluquero, y duerme mientras el peluquero le corta el pelo. Si el peluquero está ocupado cuando llega un cliente, el cliente se va a dormir en una de las otras sillas. Después de un corte de pelo, el peluquero abre la puerta de salida para el cliente y la cierra cuando el cliente se va. Si hay clientes esperando, el peluquero despierta a uno y espera que el cliente se siente. Sino, se vuelve a dormir hasta que llegue un cliente.

Los clientes y el peluquero son procesos, y la peluquería es un monitor dentro del cual los procesos interactúan. Los clientes son *clientes* que piden un servicio del peluquero, en este caso cortes de pelo. El peluquero es un *servidor* que repetidamente provee el servicio. Este tipo de interacción es un ejemplo de relación *cliente/servidor*.

Para implementar estas interacciones, podemos modelizar la peluquería con un monitor con tres procedures: *get_haircut*, *get_next_customer*, y *finished_cut*. Los clientes llaman a *get_haircut*; luego vuelven de este procedure luego de recibir un corte de pelo. El peluquero repetidamente llama a *get_next_customer* para esperar que un cliente se siente en su silla, luego le corta el pelo y finalmente llama a *finished_cut* para permitir que el cliente deje la peluquería. Se usan variables permanentes para registrar el estado de los procesos y para representar las sillas donde los procesos duermen.

Dentro del monitor, necesitamos sincronizar las acciones del peluquero y los clientes. Primero, un peluquero y un cliente necesitan *rendezvous*; es decir, el peluquero tiene que esperar que llegue un cliente, y un cliente tiene que esperar que el peluquero esté disponible. Segundo, el cliente necesita esperar hasta que el peluquero terminó de cortarle el pelo, lo cual es indicado cuando éste abre la puerta de salida. Finalmente, antes de cerrar la puerta de salida, el peluquero necesita esperar hasta que el cliente haya dejado el negocio. En resumen, tanto el peluquero como el cliente atraviesan una serie de etapas sincronizadas, comenzando con un rendezvous.

Un rendezvous es similar a una barrera de dos procesos pues ambas partes deben arribar antes de que cualquiera pueda seguir. (Sin embargo, difiere de una barrera de dos procesos pues el peluquero puede hacer rendezvous con cualquier cliente). Como con las barreras, la forma más directa de especificar las etapas de sincronización tales como estas es emplear contadores incrementales que registran el número de procesos que alcanzaron cada etapa.

Los clientes tienen dos etapas importantes: sentarse en la silla del peluquero y dejar la peluquería. Sean *cinchair* y *cleave* contadores para estas etapas. El peluquero repetidamente pasa por tres etapas: estar disponible, estar cortando el pelo, y terminar de cortar el pelo. Sean *bavail*, bbusy, y *bdone* los contadores para estas etapas. Todos los contadores son inicialmente 0. Dado que los procesos pasan secuencialmente a través de estas etapas, los contadores deben satisfacer el invariante:

C1: cinchair $\geq$ cleave $\wedge$ bavail $\geq$ bbusy $\geq$ bdone

Para asegurar que el peluquero y un cliente rendezvous antes de que el peluquero empiece a cortar el pelo del cliente, un cliente no puede sentarse en la silla del peluquero más veces que las que el peluquero estuvo disponible. Además, el peluquero no puede estar ocupado más veces que las que los clientes se sentaron en la silla. Así, también requerimos la invarianza de:

C2: cinchar $\leq$ bavail $\wedge$ bbusy $\leq$ cinchair

Finalmente, los clientes no pueden dejar la peluquería más veces que las que el peluquero terminó de cortar el pelo, lo que es expresado por el invariante:

C3: cleave $\leq$ bdone

El invariante del monitor es la conjunción de estos tres invariantes:

BARBER: C1 $\wedge$ C2 $\wedge$ C3

Necesitamos agregar estos cinco contadores a los cuerpos de los procedures de *Barber_Shop*. También necesitamos usar **await** para guardar las asignaciones a los contadores para asegurar el invariante del monitor. Finalmente, necesitamos insertar un **await** al final del *finished_cut* para asegurar que el peluquero no puede volver de *finished_cut* hasta que el cliente recién atendido dejó el negocio. El resultado es el siguiente (Este monitor, a diferencia de otros que consideramos, tiene un procedure, *get_haircut*, que contiene más de una sentencia **await**. Esto es porque un cliente pasa por dos etapas, esperando al peluquero y luego esperando que éste termine de cortarle el pelo):

```
monitor Barber_Shop       # Invariante BARBER
  var cinchar := 0, cleave := 0
  var bavail := 0, bbusy := 0, bdone := 0
  procedure get_haircut ( )       # llamado por los clientes
    await cinchair < bavail; cinchair := cinchair + 1
    await cleave < bdone; cleave := cleave + 1
  end
  procedure get_next_customer ( )     # llamado por el peluquero
    bavail := bavail + 1
    await bbusy < cinchair; bbusy := bbusy + 1
  end
  procedure finished_cut ( )            # llamado por el peluquero
    bdone := bdone + 1
    await bdone = cleave
  end     { bdone = cleave }
```

end

Aunque incrementar contadores es útil para registrar etapas a través de las cuales pasan los procesos, sus valores pueden crecer sin límite. Sin embargo, podemos evitar este problema cambiando variables. Podemos hacerlo siempre que, como aquí, la sincronización depende solo de las diferencias entre valores de los contadores. En particular, usamos una nueva variable para registrar cada diferencia que aparece en las sentencias **await**. Sean *barber*, *chair*, y *open* estas variables. Están relacionadas con los contadores como sigue:

barber = bavail - cinchair
chair = cinchair - bbusy
open = bdone - cleave

Con estas sustituciones, las nuevas variables satisfacen el invariante:

BARBER' : $0 \leq barber \leq 1 \ \wedge \ 0 \leq chair \leq 1 \ \wedge \ 0 \leq open \leq 1$

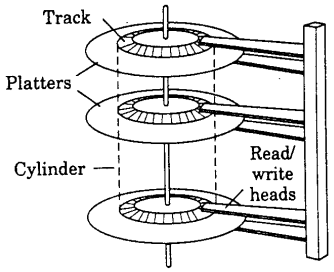
El valor de *barber* es 1 cuando el peluquero está esperando a que un cliente se siente en su silla, *chair* es 1 cuando el cliente se sentó en la silla pero el peluquero aún no está ocupado, y *open* es 1 cuando la puerta de salida fue abierta pero el cliente todavía no se fue.

Además de cambiar variables, también necesitamos implementar las sentencias **await** usando variables condición. Como en lectores/escritores, las condiciones de sincronización referencian solo variables permanentes; no referencian variables locales a los procesos. Así, usamos 4 variables condición, una para cada una de las cuatro distintas condiciones booleanas. También reemplazamos las sentencias **await** por **wait** en loops. Finalmente, agregamos sentencias **signal** en puntos donde las condiciones se hacen verdaderas. Haciendo estos cambios, tenemos la solución final:

```
monitor Barber_Shop       # Invariante BARBER'
  var barber := 0, chair := 0, open := 0
  var barber_available : cond       # signal cuando barber > 0
  var chair_occupied : cond         # signal cuando chair > 0
  var door_open : cond              # signal cuando open > 0
  var customer_left : cond          # signal cuando open = 0
  procedure get_haircut ( )       # llamado por los clientes
    do barber = 0 → wait(barber_available) od
    barber := barber - 1
    chair := chair + ; signal(chair_occupied)
    do open = 0 → wait(door_open) od
    open := open - 1; signal(customer_left)
  end
  procedure get_next_customer ( )     # llamado por el peluquero
    barber := barber + 1; signal(barber_available)
    do chair = 0 → wait(chair-occupied) od
    chair := chair - 1
  end
  procedure finished_cut ( )            # llamado por el peluquero
    open := open + 1; signal(door_open)
    do open > 0 → wait(customer_left) od
  end
end
```

SCHEDULING DE DISCO: ESTRUCTURAS DE PROGRAMA

En esta sección, examinamos otro problema importante (hacer scheduling del acceso a una cabeza de disco movible) y mostramos cómo pueden aplicarse las técnicas descriptas en la sección previa para desarrollar una solución. También examinamos dos maneras distintas en la cual puede estructurarse la solución. El problema de scheduling de disco es representativo de numerosos problemas de scheduling. También, cada una de las estructuras de solución es aplicable en otras numerosas situaciones. Comenzamos resumiendo las características de los discos de cabeza movible, usados para almacenar datos.



El disco contiene varios “platos” conectados a un eje central y que rotan a velocidad constante. Los datos son almacenados sobre las superficies de los platos. Cada plato es como un fonógrafo, excepto que las pistas forman círculos concéntricos separados en lugar de estar conectados en espiral. Las pistas en la misma posición relativa sobre distintos platos forman un cilindro. Los datos son accedidos posicionando una cabeza lectora/escritora sobre la pista apropiada, luego esperar que el plato rote hasta que el dato deseado pase por la cabeza. Normalmente, hay una cabeza lectora/escritora por plato. Estas cabezas están conectadas a un único brazo, el cual puede moverse hacia adentro y afuera de modo que las cabezas pueden ubicarse en cualquier cilindro y por lo tanto sobre cualquier pista.

La dirección física de cualquier porción de datos almacenada en el disco consta de un cilindro, un número de pista, la cual especifica el plato, y un desplazamiento, el cual especifica la distancia desde un punto fijo que es el mismo en cada pista. Para acceder al disco, un programa ejecuta una instrucción de entrada/salida específica. Los parámetros a tal instrucción son una dirección física del disco, un contador del número de bytes a transferir, una indicación de la clase de transferencia a realizar (lectura o escritura), y la dirección de un buffer que contiene los datos a escribir o donde se pondrán los datos leídos.

Como consecuencia de estas características, el tiempo de acceso al disco depende de tres cantidades: *seek time* para mover una cabeza lectora/escritora al cilindro apropiado, *rotational delay*, y el *transmission time* de datos. El tiempo de transmisión depende solo del número de bytes a ser transferido, pero las otras dos cantidades dependen del estado del disco. En el mejor caso, un cabeza ya está en el cilindro pedido, y el área de pista pedida está justo comenzando a pasar bajo la cabeza. En el peor caso, las cabezas deben ser movidas atravesando el disco, y la pista pedida tiene que hacer una revolución completa. Una característica de los discos de cabeza movible es que el tiempo requerido para mover las cabezas de un cilindro a otro es una función creciente de la distancia entre los dos cilindros. También, el tiempo que toma mover una cabeza de un cilindro a otro es mucho mayor que el tiempo de rotación del plato. Así, la manera más efectiva de reducir el tiempo de acceso promedio es minimizar el movimiento de la cabeza y por lo tanto reducir el tiempo de seek. (También ayuda reducir el delay rotacional, pero esto es más difícil de realizar pues las demoras rotacionales son típicamente bastante cortas).

Asumimos que hay varios clientes que usan el disco. Por ejemplo, en un SO multiprogramado, podrían ser procesos que ejecutan comandos del usuario o procesos del sistema que implementan un manejo de memoria virtual. Siempre que un solo cliente a la vez quiere acceder al disco, no se puede ganar nada no permitiendo que el disco sea accedido inmediatamente pues en general no podemos saber cuándo otro cliente también puede querer acceder el disco. Así, el scheduling de disco se aplica solo si dos o más clientes quieren acceder el disco.

Una estrategia de scheduling atractiva es siempre seleccionar el pedido pendiente que quiere acceder el cilindro más cercano al que están posicionadas las cabezas actualmente. Esta es llamada estrategia *shortest-seek-time* (SST) pues minimiza el tiempo de seek. Sin embargo, SST es unfair pues una corriente de pedidos por cilindros cercanos a la posición corriente podría impedir que se satisfagan otros pedidos. Aunque tal inanición es altamente improbable, no hay límite sobre cuánto podría ser demorado un pedido.

Una estrategia fair de scheduling alternativa es mover las cabezas en una sola dirección hasta que todos los pedidos en esa dirección fueron servidos. En particular, seleccionar el pedido más cercano a la posición corriente de la cabeza en la dirección en que las cabezas se movieron. Si no hay pedidos pendientes en la dirección corriente, se invierte la dirección. Esta estrategia es llamada SCAN, LOOK, o *algoritmo del ascensor* pues es análogo a la manera en que un ascensor visita los pisos, levantando y bajando clientes. El único problema con esta estrategia es que un pedido pendiente justo detrás de la posición actual de la cabeza no será servido hasta que la cabeza llegue al final y vuelva. Esto lleva a una larga varianza en el tiempo de espera antes de que un pedido sea servido.

Una tercera estrategia s similar a la segunda pero reduce la varianza del tiempo de espera. En esta estrategia, llamada CSCAN o CLOOK (C por circular), los pedidos son servidos en una sola dirección, por ejemplo, del cilindro más externo al más interno. En particular, hay una sola dirección de búsqueda, y el pedido más cercano a la posición actual de la cabeza en esa dirección se selecciona. Cuando no hay más pedidos más allá de la posición corriente, la búsqueda comienza nuevamente del cilindro más externo. Esto es análogo a un ascensor que solo toma pasajeros hacia arriba. CSCAN es casi tan eficiente como el algoritmo del ascensor con respecto a reducir el tiempo de seek pues en la mayoría de los discos toma solo aproximadamente dos veces mover las cabezas a través de todos los cilindros que lo que toma moverlas de un cilindro al próximo. Más aún, la estrategia CSCAN es fair siempre que una corriente continua de pedidos por la posición corriente no se permite que provoque inanición sobre otros pedidos pendientes.

En el resto de la sección, desarrollamos dos soluciones estructuralmente distintas al problema. En la primera, el scheduler es implementado por un monitor distinto, como en la solución a lectores/escritores. En la segunda, el scheduler es implementado por un monitor que actúa como intermediario entre los usuarios del disco y un proceso que realiza los accesos reales al disco; esta estructura es similar a la de la solución del sleeping barber. Ambos monitores implementan la estrategia CSCAN, pero pueden ser fácilmente modificados para implementar cualquiera de las estrategias de scheduling.

Scheduling como un Monitor Separado

Una manera de organizar una solución al problema de scheduling de disco es tener el scheduler como un monitor separado del recurso que es controlado, en este caso, el disco. La solución tiene tres clases de componentes: procesos usuario, el scheduler, y los procedures o procesos que realizan las trasferencias de disco. El scheduler es implementado por un monitor para que los datos de scheduling sean accedidos solo por un proceso usuario a la vez. El monitor provee dos operaciones: *request* y *release*.

Un proceso usuario que quiere acceder al cilindro *cyl* primero llama a *request*(*cyl*); el proceso retorna de *request* cuando el scheduler seleccionó su pedido. El proceso usuario luego accede al disco, por ejemplo, llamando a un procedure o comunicándose con un proceso manejador del disco. Luego de acceder al disco, el usuario llama a *release* para que otro pedido pueda ser seleccionado. Entonces la interfase al scheduler es:

```
Disk_Scheduler.request(cyl)
Accede al disco
Disk_Scheduler.release( )
```

Disk_Scheduler tiene el doble rol de hacer el scheduling de los pedidos y asegurar que a lo sumo un proceso a la vez usa el disco. Así, todos los usuarios *deben* seguir este protocolo.

Asumimos que los cilindros están numerados de 0 a MAXCYL y que el scheduling es emplear la estrategia CSCAN con una dirección de búsqueda de 0 a MAXCYL. Como es usual, el paso crítico al derivar una solución correcta es establecer precisamente las propiedades que debe tener la solución. Aquí, a lo sumo un proceso a la vez puede tener permiso para usar el disco, y los pedidos pendientes son servidos en orden CSCAN.

Sea que *position* indique la posición corriente de la cabeza, es decir, el cilindro que está siendo accedido por el proceso que está usando el disco. Cuando el disco no está siendo accedido, *position* será -1.

Para implementar scheduling CSCAN, necesitamos distinguir entre los pedidos pendientes a ser servidos en el scan corriente a través del disco y los que serán servidos en el próximo scan. Sean *C* y *N* conjuntos disjuntos que contienen estos pedidos. En particular, los contenidos de *C* y *N* son:

C: { cyl_i | $cyl_i \geq$ position }
N: { cyl_i | $cyl_i \leq$ position }

donde los cyl_i son los argumentos de los pedidos pendientes (Dado que más de un proceso podría requerir el mismo cilindro, los distintos valores de cyl_i necesitan ser distinguibles. Aquí, subscribimos *cyl* por el índice del proceso usuario *i*).

Un pedido pendiente para el cual cyl_i = position es almacenado solo en uno de los conjuntos. Dado que la estrategia CSCAN sirve los pedidos en orden ascendente, ambos conjuntos están ordenados por valor creciente de cyl_i; los pedidos por el mismo cilindro son ordenados por tiempo de inserción. *C*, *N*, y *position* son las variables permanentes del monitor. Para un scheduling correcto, deben satisfacer el invariante:

DISK: C y N son conjuntos ordenados $\wedge$
($\forall$ cyl_i : $cyl_i \in$ C : position $\leq cyl_i$) $\wedge$
($\forall$ cyl_j : $cyl_j \in$ N : position $\geq cyl_j$) $\wedge$
(position = -1) $\Rightarrow$ (C = $\varnothing$ $\wedge$ N = $\varnothing$)

Cuando un proceso P_i llama a *request*, toma una de tres acciones. Si *position* es -1, el disco está libre; así, el proceso setea *position* a cyl_i y accede al disco. Si el disco no está libre y cyl_i > *position*, el proceso inserta cyl_i en el conjunto *C*; en otro caso, inserta cyl_i en *N*. Usamos *N* en lugar de *C* cuando cyl_i = position para evitar potencial unfairness. Luego de registrar cyl_i en el conjunto apropiado, el proceso se demora hasta que se le da el acceso al disco, es decir, hasta que *position* = cyl_i.

Cuando un proceso llama a *release*, actualiza las variables permanentes para mantener DISK. Si *C* no está vacío, hay pedidos pendientes para el scan corriente. En este caso, el proceso que libera remueve el primer elemento de *C* y setea *position* a su valor. Si *C* está vacío pero *N* no, necesitamos comenzar el próximo scan; es decir, necesita convertirse en el scan corriente. El proceso que libera hace esto swapeando *C* y *N* (lo cual seta *N* al conjunto vacío), removiendo el primer elemento de *C*, y seteando *position* a su valor. Finalmente, si ambos conjuntos están vacíos, el proceso que libera setea *position* a -1 para indicar que el disco está libre.

Juntando todo esto, tenemos el siguiente monitor (aquí, *insert*(*conjunto*,*valor*) pone *valor* en la posición apropiada en el *conjunto* ordenado; *delete*(*conjunto*) remueve el primer elemento del *conjunto* y retorna su valor):

```
monitor Disk_Scheduler          # Invariante DISK
  var position : int := -1
  var C, N : ordered set of int := ∅
  procedure request (cyl_i : int )
    if position = -1 → position := cyl_i
    □ position ≠ -1 and cyl_i > position →
            insert(C,cyl_i)
            await position = cyl_i
    □ position ≠ -1 and cyl_i ≤ position →
```

```
            insert(N,cyl_i)
            await position = cyl_i
  fi
 end
 procedure release ( )
  if C ≠ ∅ → position := delete(C)
  □ C = ∅ and N ≠ ∅ → C := N; position := delete(C)
  □ C = ∅ and N = ∅ → position := -1
  fi
 end
end
```

El paso final de derivación de una solución es implementar los **await** usando variables condición. Tenemos una situación en la cual hay un orden estático entre las condiciones de espera, y entonces es posible usar **wait** con prioridad para implementar cada **await**. En particular, los pedidos en los conjuntos *C* y *N* son servidos en orden ascendente de cyl_i. También tenemos una situación como en el semáforo FIFO: cuando el disco es liberado, el permiso para acceder al disco es transferido a un proceso específico. En particular, seteamos *position* al valor del pedido pendiente que será el próximo en ser servido. Por estos dos atributos, podemos implementar **await** eficientemente.

Para representar los conjuntos *C* y *N*, sea *scan*[0:1] un arreglo de variables condición indexado por enteros *c* y *n*. Cuando un proceso que pide necesita insertar su parámetro *cyl* en *C* y luego esperar a que *position* sea igual a *cyl*, simplemente ejecuta **wait**(*scan*[*c*], *cyl*). Análogamente, un proceso inserta su pedido en el conjunto *N* y luego se demora ejecutando **wait**(*scan*[*n*], *cyl*). Dado que las variables condición son colas ordenadas y solo un proceso es despertado cuando se ejecuta **signal**, ya no es necesario distinguir entre valores iguales de *cyl*.

También necesitamos cambiar el cuerpo de *release* para usar esta representación de los conjuntos *C* y *N*. En particular, usamos **empty** para testear si un conjunto está vacío, usamos **minrank** para determinar su valor más chico, y usamos **signal** para remover el primer elemento y al mismo tiempo despertar al proceso apropiado. También, swapeamos los conjuntos cuando es necesario simplemente intercambiando los valores de *c* y *n* (por eso *scan* es un arreglo).

Incorporando estos cambios, tenemos la siguiente solución final:

```
monitor Disk_Scheduler          # Invariante DISK
 var position := -1, c := 0, n := 1
 var scan[0:1] : cond     # scan[c] es señalizado cuando el disco es liberado
 procedure request (cyl : int )
  if position = -1 → position := cyl
  □ position ≠ -1 and cyl > position → wait(scan[c],cyl)
  □ position ≠ -1 and cyl ≤ position → wait(scan[n],cyl)
  fi
 end
 procedure release ( )
  if not empty(scan[c]) → position := minrank(scan[c])
  □ empty(scan[c]) and not empty(scan[n]) →
            c := n; position := minrank(scan[c])
  □ empty(scan[c]) and empty(scan[n]) → position := -1
  fi
 end
end
```

Dado que *c* es el índice del scan corriente al final de *release*, es suficiente incluir solo una sentencia **signal**. Si de hecho *position* es -1 en este punto, *scan*[*c*] estará vacío, y así **signal** no tendrá efecto.

Los problemas de scheduling tomo este están entre los más difíciles de resolver correctamente, cualquiera sea el mecanismo de sincronización empleado. El paso crítico es especificar exactamente el orden en el cual los procesos serán servidos. Cuando el orden de servicio relativo es estático (como aquí) podemos usar conjuntos ordenados. Esto lleva a una solución que puede hacer uso de sentencias **wait** con prioridad. Desafortunadamente, el orden de servicio relativo no es siempre estático; por ejemplo, un proceso que es demorado un tiempo largo podría incrementar su prioridad para evitar inanición. Implementar una estrategia de scheduling dinámica requiere por ejemplo usar variables condición privadas para despertar procesos individuales.

Scheduler como un Intermediario

Implementar *Disk_Scheduler* (o cualquier controlador de recurso) como un monitor separado es una manera viable de estructurar una solución a un problema de scheduling/alocación. Dado que el scheduler está separado, puede ser diseñado independientemente de las otras componentes. Sin embargo, esta separación introduce dos problemas potenciales:

- La presencia del scheduler es visible a los procesos que usan el disco; si el scheduler es borrado, los procesos de usuario cambian
- Todos los procesos usuario deben seguir el protocolo requerido de pedir el disco, usarlo y luego liberarlo. Si alguno falla al ejecutar este protocolo, el scheduling se arruina; además, podría no estar asegurado el acceso exclusivo al disco.

Ambos problemas pueden ser aliviados si el protocolo de uso del disco está embebido en un procedure y los procesos usuario no acceden directamente ni al disco ni al scheduler del disco. Sin embargo, esto introduce otra capa de procedures y algo de ineficiencia.

Además de los dos problemas anteriores, hay un tercero si el disco es accedido por un proceso manejador del disco en lugar de por procedures llamados directamente por los procesos usuarios. En particular, después de ser otorgado el acceso al disco, un proceso usuario debe comunicarse con el manejador para pasarle argumentos y recibir resultados. Estas comunicaciones podrían ser implementadas por dos instancias del monitor de buffer limitado. Pero la interfase de usuario consistiría entonces de tres monitores (el scheduler y dos buffers limitados) y el usuario tendría que hacer un total de 4 llamados al monitor cada vez que usa el dispositivo. Dado que los usuarios del disco y el disk driver tienen una relación cliente/servidor, podríamos implementar la interfase de comunicación usando una variante de la solución al problema del sleeping barber. Pero aún tendríamos dos monitores (uno para scheduling y uno para interacción entre procesos usuario y el disk driver).

Cuando un disco es controlado por un proceso driver, la mejor aproximación posible es combinar el scheduler y la interfase de comunicación en un único monitor. Esencialmente, el scheduler se convierte en intermediario entre los procesos usuario y el disk driver. El monitor transmite los pedidos de usuario al driver en el orden de preferencia deseado. El resultado neto es que la interfase del disco emplea solo un monitor, y el usuario debe hacer solo un llamado al monitor por acceso al disco. Además, la presencia o ausencia de scheduling es transparente. Más aún, no hay un protocolo de varios pasos que el usuario puede fallar al cumplir. Así, esta aproximación evita las tres dificultades causadas por el scheduler como monitor separado.

En el resto de esta sección, mostramos cómo transformar la solución al problema del sleeping barber en una interfase manejadora de disco que provee comunicación entre clientes y el disk driver e implementa scheduling CSCAN. Necesitamos hacer varios cambios a la solución del sleeping barber. Primero, necesitamos renombrar los procesos, monitor y procedures. Segundo, parametrizar los procedures para transferir los pedidos desde los usuarios (clientes) al disk driver (peluquero) y para transferir los resultados de vuelta; en esencia, hay que cambiar la “silla del peluquero” y la “puerta de salida” por buffers de comunicación. Finalmente, necesitamos agregar scheduling al rendezvous usuario/disk driver para que el driver sirva al pedido de usuario preferido. Estos cambios dan una interfase de disco con el siguiente outline:

```
monitor Disk_Interface
  variables permanentes para estado, scheduling y transferencia de datos
```

```
 procedure use_disk( cyl : int, parámetros de transferencia y resultado )
  Esperar turno para usar el driver
  Almacenar los parámetros de transferencia en variables permanentes
  Esperar a que se complete la transferencia
  Recuperar los resultados desde las variables permanentes
 end
 procedure get_next_request( res parámetros de transferencia )
  Seleccionar próximo pedido
  Esperar que los parámetros de transferencia sean almacenados
 end
 procedure finished_transfer( resultados )
  Almacenar resultados en variables permanentes
  Esperar que los resultados sean recuperados por el cliente
 end
end
```

Para refinar este outline en una solución real, empleamos la misma sincronización básica que en el sleeping barber. Sin embargo, agregamos scheduling como en el monitor *Disk_Scheduler* y pasaje de parámetros como en un buffer de un slot. Esencialmente, el invariante del monitor para *Disk_Interface* se convierte en la conjunción del invariante BARBER', el invariante DISK, y el invariante BB del buffer limitado simplificado al caso de un buffer de un slot.

Un proceso usuario espera su turno para acceder al disco ejecutando las mismas acciones que en el procedure *request* del monitor *Disk_Scheduler*. Sin embargo, inicialmente, setearemos *position* a -2 para indicar que el disco ni está disponible ni en uso hasta después de que el driver hace su primer llamado a *get_next_request*; por lo tanto, los usuarios necesitan esperar el primer scan para comenzar.

Cuando es el turno de un usuario para acceder al disco, el proceso usuario deposita sus argumentos de transferencia en variables permanentes, luego espera para buscar los resultados. Después de seleccionar el próximo pedido de usuario, el proceso driver espera para buscar los argumentos de transferencia del usuario. El driver luego realiza la transferencia a disco pedida. Cuando está terminada, el driver deposita los resultados y luego espera que sean buscados. Los depósitos y búsquedas son implementados como para un buffer de un slot. Estos refinamientos llevan al siguiente monitor:

```
monitor Disk_Interface        # Invariante combina BARBER', DISK, y BB
 var position := -2, c := 0, n := 1, scan[0:1] : cond
 var arg_area : arg_type, result_area : result_type, args := 0, results := 0
 var args_stored, results_stored, results_retrieved : cond
 procedure use_disk(cyl : int, parámetros de transferencia y resultado)
  if position = -1 → position := cyl
  □ position ≠ -1 and cyl > position → wait(scan[c],cyl)
  □ position ≠ -1 and cyl ≤ position → wait(scan[n],cyl)
  fi
  arg_area := parámetros de transferencia
  args := args + 1; signal(args_stored)
  do results = 0 → wait(results_stored) od
  parámetros resultado := result_area
  results := results - 1; signal(results_retrieved)
 end
 procedure get_next_request ( res parámetros de transferencia )
  if not empty(scan[c]) → position := minrank(scan[c])
  □ empty(scan[c]) and not empty(scan[n]) →
              c := n; position := minrank(scan[c])
  □ empty(scan[c]) and empty(scan[n]) → position := -1
  fi
  signal(scan[c])
  do args = 0 → wait(args_stored) od
  parámetros de transferencia := arg_area; args := args - 1
 end
```

```
  procedure finished_transfer( parámetros resultado )
    result_area := parámetros resultado; results := results + 1
    signal(results_stored)
    do results > 0 → wait(results_retrieved) od
  end
end
```

Aunque esta interfase usuario/disk-driver es bastante eficiente, puede ser aún más eficiente con dos cambios relativamente simples. Primero, el disk driver puede empezar sirviendo al próximo usuario más rápido si *finished_transfer* es modificado para que el driver no espere que los resultados de la transferencia previa hayan sido recuperados. Sin embargo, debemos ser cuidadosos para asegurar que el área de resultados no sea sobreescrita en el caso de que el driver complete otra transferencia antes de que los resultados de la anterior hayan sido recuperados. El segundo cambio es combinar los dos procedures llamados por el disk driver. Esto elimina un llamado al monitor por acceso al disco. Implementar este cambio requiere modificar levemente la inicialización de *results*.

APROXIMACIONES ALTERNATIVAS A SINCRONIZACION

Los monitores, como los hemos definido, tienen tres características distinguibles. Primero, un monitor encapsula sus variables permanentes; tales variables pueden ser accedidas solo llamando a un procedure del monitor. Segundo, los procedures del monitores ejecutan con exclusión mutua. Estas dos características implican que las variables permanentes no están sujetas a acceso concurrente. Tercero, la sincronización por condición es provista por medio de variables condición y las operaciones **wait** y **signal**. La ejecución de **wait** demora al proceso ejecutante y libera temporariamente la exclusión del monitor; esta es la única manera (además de retornar de un procedure del monitor) en la cual un proceso libera la exclusión. La ejecución de **signal** despierta un proceso demorado, si hay, pero el proceso que señaliza mantiene el acceso exclusivo al monitor; el proceso despertado reanuda la ejecución en algún momento futuro cuando pueda readquirir el acceso exclusivo.

Monitores con estas características fueron incluidos en el lenguaje de programación Mesa, y esta aproximación para sincronización por condición se usa también dentro del SO Unix. Sin embargo, hay otras maneras para proveer sincronización por condición. Además, la exclusión mutua no siempre se requiere dentro de los monitores. En esta sección, describimos y comparamos mecanismos alternativos para sincronización por condición.

Disciplinas de Señalización Alternativas

En esta sección, asumimos que los procedures del monitor ejecutan con exclusión mutua, y enfocamos distintas maneras en que puede realizar la sincronización por condición. Ya vimos dos técnicas para especificar tal sincronización. Al derivar monitores, primero usamos **await** para demorar la ejecución cuando es necesario. Luego implementamos los **await** usando variables condición y operaciones **wait** y **signal**. De hecho, **await** podría ser usada directamente para especificar sincronización, evitando el último paso de derivación a expensas de una implementación subyacente más costosa.

Estas dos aproximaciones son similares en que los puntos de demora son programados explícitamente. Difieren en la manera de hacer la señalización. Con **await**, la señalización es implícita; es provista por un kernel o por código generado por el compilador. Por esta razón, **await** es llamado mecanismo de *automatic signaling* (AS). Por contraste, con variables condición, la señalización es explícita. Más aún, **signal**, como lo definimos, es *no preemptivo* pues el proceso que lo realiza continúa ejecutando. Otro proceso no obtiene el control del monitor hasta que el señalador espere o retorne. Esta clase de señalización es llamada *signal and continue* (SC).

En los lenguajes de programación se incorporaron otros tres mecanismos para sincronización por condición en monitores. Todos emplean variables condición y operaciones **wait** y **signal**, pero dan distintas semánticas a **signal**. En los tres casos, **signal** es *preemptivo*: Si un proceso ejecuta **signal** y despierta a otro, el proceso señalador es forzado a demorar, y el despertado es el próximo en ejecutar en el monitor. Así, el permiso para ejecutar en el monitor es transferido del señalador al despertado, el cual podría pasar el permiso a otro proceso, etc. La razón para la señalización preemptiva es que la condición señalizada se garantiza que es true en el momento que el proceso despertado reanuda la ejecución, como en el método de passing the baton al usar semáforos. Por contraste, con SC o el señalador o algún otro proceso podría invalidar la condición antes de que el proceso señalizado reanude la ejecución.

Los tres mecanismos de señalización preemptivos difieren con respecto a qué sucede con el proceso señalador. La primera posibilidad es que el proceso señalador sea forzado a salir del monitor, es decir, a retornar del procedure que está ejecutando. Esto es llamado *signal and exit* (SX) o signal and return; fue incluido en Pascal Concurrente. Otra posibilidad es que el proceso señalador pueda seguir ejecutando en el monitor, pero primero debe esperar hasta readquirir el acceso exclusivo. Esto es llamado *signal and wait* (SW) y fue incluido en Modula y Euclid Concurrente. La posibilidad final es una variación de SW: el proceso señalador es forzado a esperar, pero se le asegura que tomará el control del monitor antes que cualquier nuevo llamado a los procedures del monitor. Esta disciplina es llamada *signal and urgent wait* (SU) y fue incluida en Pascal-Plus.

Por lo tanto, hay cinco disciplinas distintas de señalización:

- Automatic Signaling (AS): implícito y no preemptivo
- Signal and Continue (SC): explícito y no preemptivo
- Signal and Exit (SX): explícito, preemptivo, y el señalador sale del monitor
- Signal and Wait (SW): explícito, preemptivo, y el señalador espera
- Signal and Urgent Wait (SU): explícito, preemptivo, y el señalador espera, pero ejecuta nuevamente antes que nuevos entries al monitor

Aunque parecen bastante diferentes, cada disciplina puede usarse para simular las otras. Así pueden resolver los mismos problemas, aunque SX puede requerir cambiar la interfase a un monitor. Más aún, en muchos casos las disciplinas de señalización explícitas pueden ser usadas intercambiablemente. Las disciplinas difieren principalmente con respecto a su facilidad de uso para desarrollar soluciones correctas y con respecto al costo de implementación.

PASAJE DE MENSAJES

Las construcciones de sincronización que utilizamos hasta ahora están basadas en variables compartidas, por lo tanto se usan en programas concurrentes que ejecutan en hardware con procesadores que comparten memoria. Sin embargo, las *arquitecturas de red* en las que los procesadores comparten solo una red de comunicaciones son cada vez más comunes (por ej. redes de WS o multicomputadores como hipercubos). También se emplean combinaciones híbridas de memoria compartida y arquitecturas de red. Aún en arquitecturas de memoria compartida puede ser necesario o conveniente que los procesos no compartan variables.

Para escribir programas para una arquitectura de red, primero es necesario definir la interfase de red, es decir, las operaciones de red primitivas. Podrían ser simplemente operaciones de read y write análogas a las de variables compartidas. Pero los procesos deberían usar sincronización busy-waiting. Una aproximación mejor es definir operaciones especiales que incluyan sincronización (como las operaciones sobre semáforos son operaciones especiales sobre variables compartidas). Tales operaciones de red se llaman *primitivas de pasaje de mensajes*. De hecho, el pasaje de mensajes (MP) puede verse como una extensión de los semáforos para transportar datos y proveer sincronización.

Con MP los procesos comparten *canales*. Un canal es una abstracción de una red de comunicación física; provee un path de comunicación entre procesos. Los canales son accedidos por dos clases de primitivas: **send** y **receive**. Para iniciar una comunicación, un proceso envía un mensaje a un canal; otro proceso toma el mensaje recibiéndolo por el canal. La comunicación se lleva a cabo pues los datos fluyen del emisor al receptor. La sincronización está dada en que un mensaje no puede ser recibido hasta después de haber sido enviado.

Cuando se usa MP, los canales normalmente son los únicos objetos que comparten los procesos. Así, toda variables es local a y accesible por un proceso (su *cuidador*). Esto implica que las variables no están sujetas a acceso concurrente, y por lo tanto no se requieren mecanismos para exclusión mutua. La ausencia de variables compartidas también cambia la manera en la cual se programa la sincronización por condición pues sólo un proceso cuidador puede examinar las variables que codifican una condición. Esto requiere usar técnicas de programación distintas a las usadas con VC. La consecuencia final de la ausencia de VC es que los procesos no necesitan ejecutar en procesadores que comparten memoria; en particular, los procesos pueden estar distribuidos entre procesadores. Por esta razón, los programas concurrentes que usan MP son llamados *programas distribuidos*. Sin embargo tales programas pueden ser ejecutados en procesadores centralizados: en este caso, los canales son implementados usando memoria compartida en lugar de una red de comunicación.

Todas las notaciones de programación basadas en MP proveen canales y primitivas para enviar y recibir de ellos. Varían en la manera en que se proveen y nombran los canales, la manera en que se usan y la manera en que se sincroniza la comunicación. Por ejemplo, los canales pueden ser globales a los procesos, estar conectados a receptores, o estar conectados a un único emisor y receptor. También pueden proveer flujo de información uni o bidireccional. O pueden ser sincrónicos o asincrónicos.

Se describirán las 4 combinaciones generales de estas elecciones de diseño que son más populares. Cada una es especialmente adecuada para resolver algunos problemas, y puede ser implementada con razonable eficiencia.

El cap. 7 examina una notación en la cual los canales tienen capacidad ilimitada, por lo que la primitiva de **send** no causa el bloqueo de un proceso (*pasaje de mensajes asincrónico*, AMP).

El cap. 8 da una notación en la cual la comunicación y la sincronización están muy asociadas. En particular, cada canal provee un link directo entre dos procesos, y un proceso que envía un mensaje se demora hasta que el otro lo reciba (*pasaje de mensajes sincrónico*, SMP).

El cap. 9 examina dos notaciones adicionales: remote procedure call (RPC) y rendezvous. Estos combinan aspectos de monitores y SMP. Como con monitores, un módulo o proceso exporta operaciones y las operaciones son invocadas por una sentencia **call**. Como con SMP, la ejecución de **call** es sincrónica (el proceso llamador se demora hasta que la invocación fue servida y se retornaron resultados). Así una operación es un canal de comunicaciones bidireccional. Una invocación es servida en una de dos maneras. Una aproximación es crear un nuevo proceso: esto es llamado RPC pues el proceso que sirve es declarado como un procedure y podría ejecutar en un procesador distinto que el proceso que lo llamó. La otra forma es *rendezvous* con un proceso existente. Un rendezvous es servido por medio de una sentencia de entrada (o accept) que espera una invocación, la procesa y luego retorna resultados.

Como veremos, las 4 aproximaciones son equivalentes en el sentido de que un programa escrito usando un conjunto de primitivas puede ser reescrito usando cualquiera de los otros. Sin embargo, cada aproximación es mejor para resolver determinados tipos de problemas.

Cuando los procesos interactúan usando VC, cada proceso accede directamente las variables que necesita. Por lo tanto hasta ahora nuestro principal objetivo fue sincronizar el acceso a variables compartidas y proveer exclusión mutua y sincronización por condición. Por contraste, con MP, solo los canales son compartidos, de modo que los procesos se deben comunicar para interactuar. Por lo tanto el principal objetivo será sincronizar la comunicación entre procesos: cómo se hace depende de la manera en que interactúan los procesos.

Hay 4 clases básicas de procesos en un programa distribuido: filtros, clientes, servidores y pares (peers). Un *filtro* es un transformador de datos: recibe streams de valores de datos desde sus canales de entrada , realiza alguna computación sobre esos valores, y envía streams de resultados a sus canales de salida. A causa de estos atributos, podemos diseñar un filtro independiente de otros procesos. Más aún, fácilmente podemos conectar filtros en redes que realizan computaciones más grandes: lo único que se requiere es que cada filtro produzca salidas que cumplan las suposiciones de entrada de los filtros que consumen esas salidas.

Un *cliente* es un proceso triggering (disparador); un *servidor* es un proceso reactivo. Los cliente hacen pedidos que disparan reacciones de los servidores. Un cliente inicia la actividad, a veces de su elección; con frecuencia luego se demora hasta que su pedido fue servido. Un servidor espera que le hagan pedidos, luego reacciona a ellos. La acción específica que toma un servidor puede depender de la clase de pedido, los parámetros del mensaje de pedido, y el estado del servidor; el servidor podría ser capaz de responder a un pedido inmediatamente, o podría tener que salvar el pedido y responder luego. Un servidor con frecuencia es un proceso que no termina y provee servicio a más de un cliente. Por ejemplo, un file server en un sistema distribuido maneja un conjunto de archivos y pedidos de servicios de cualquier cliente que quiere acceder a esos archivos.

Un *peer* es uno de un conjunto de procesos idénticos que interactúan para proveer un servicio o para resolver un problema. Por ejemplo, dos *peers* podrían manejar cada uno una copia de un archivo replicado e interactuar para mantenerlas consistentes. O varios peers podrían interactuar para resolver un problema de programación paralela, resolviendo cada uno una parte del problema.

Los ejemplos ilustrarán distintas clases de filtros, clientes, servidores y peers. También introducen varios patrones de interacción entre procesos. Cada *paradigma de interacción* es un ejemplo o modelo de un patrón de comunicación y la técnica de programación asociada que puede usarse para resolver una variedad de problemas de programación distribuida interesantes. Dado que cada paradigma es distinto, no hay un único método de derivación para programas distribuidos. Sin embargo, cada proceso es diseñado en forma similar a un programa secuencial. Más aún, los mismos conceptos clave que se aplican a programas de variables compartidas también se aplican a programas basados en mensajes. En particular, nuevamente usaremos invariantes para caracterizar tanto los estados de procesos individuales como el estado global mantenido por un conjunto de procesos. Cuando sea necesario, también guardaremos las acciones de comunicación de los procesos para asegurar que se mantienen los invariantes. Dado que cada proceso tiene acceso directo solo a sus propias variables, uno de los desafíos al diseñar programas distribuidos es mantener (o determinar) un estado global.

Pasaje de Mensajes Asincrónico

Con AMP, los canales de comunicación son colas ilimitadas de mensajes. Un proceso agrega un mensaje al final de la cola de un canal ejecutando una sentencia **send**. Dado que la cola es (conceptualmente) ilimitada, la ejecución de **send** no bloquea al emisor. Un proceso recibe un mensaje desde un canal ejecutando una sentencia **receive**. La ejecución de **receive** demora al receptor hasta que el canal esté no vacío; luego el mensaje al frente del canal es removido y almacenado en variables locales al receptor.

Los canales son como semáforos que llevan datos. Por lo tanto, las primitivas **send** y **receive** son como las operaciones **V** y **P**. De hecho, si un canal contiene solo mensajes nulos (es decir, sin ningún dato) entonces **send** y **receive** son exactamente como **V** y **P**, con el número de "mensajes" encolados siendo el valor del semáforo.

Las primitivas de AMP fueron incluidas en varios lenguajes de programación; también son provistas por varios SO.

Las próximas secciones examinan distintas maneras en las cuales grupos de procesos pueden interactuar en programas distribuidos. Se proveen un número de técnicas útiles para diseñar programas distribuidos. Describimos los siguiente paradigmas de interacción:

- flujo de datos en una dirección en redes de filtros
- pedidos y respuestas entre clientes y servidores
- interacción back-and-forth (heartbeat) entre procesos vecinos
- broadcasts entre procesos en grafos completos
- token-passing por las aristas en un grafo
- coordinación entre procesos servidores descentralizados
- workers replicados compartiendo una bolsa de tareas

Ilustramos los paradigmas resolviendo una variedad de problemas, incluyendo sorting paralelo, disk scheduling, computación de la topología de una red, detección de terminación distribuida, archivos replicados, y cuadratura adaptiva paralela.

NOTACION

Se propusieron muchas notaciones distintas para AMP. Aquí, empleamos una que es representativa y simple.

Con AMP, un canal es una cola de mensajes que fueron enviados pero aún no recibidos. Una declaración de canal tiene la forma:

chan *ch*(*id*$_1$: *tipo*$_1$, ... , *id*$_n$: *tipo*$_n$)

El identificador *ch* es el nombre del canal. Los *id*$_i$ y *tipo*$_i$ son los nombre y tipos de los campos de datos de los mensajes transmitidos vía el canal. Los nombres de los campos son opcionales; serán usados cuando ayude a documentar lo que representa cada campo. Por ejemplo, lo siguiente declara dos canales:

chan input(**char**)
chan disk_access(cylinder,block,count : **int**; buffer : **ptr**[*]**char**)

El canal *input* se usa para transmitir mensajes de un solo carácter. El canal *disk_access* contiene mensajes con 4 campos. En muchos ejemplos usaremos arreglos de canales:

chan result[1:n](**int**)

Un proceso envía un mensaje al canal *ch* ejecutando:

send *ch*(*expr*$_1$, ... , *expr*$_n$)

Las *expr*$_i$ son expresiones cuyos tipos deben ser los mismos que los de los campos correspondientes en la declaración de *ch*. El efecto de ejecutar **send** es evaluar las expresiones, luego agregar un mensaje conteniendo esos valores al final de la cola asociada con el canal *ch*. Dado que esta cola es conceptualmente ilimitada, la ejecución de **send** nunca causa demora; por lo tanto es una primitiva *no bloqueante*.

Un proceso recibe un mensaje desde el canal *ch* ejecutando:

receive *ch*(*var*$_1$, ... , *var*$_n$)

Las *var*$_i$ son variables cuyos tipos deben ser los mismos que los de los campos correspondientes en la declaración de *ch*. El efecto de ejecutar **receive** es demorar al receptor hasta que haya al menos un mensaje en la cola del canal. Luego el mensaje al frente de la cola es removido, y sus campos son asignados a las *var*$_i$. Así, a diferencia de **send**, **receive** es una primitiva *bloqueante* pues podría causar demora. La primitiva **receive** tiene semántica bloqueante de manera que el proceso receptor no tiene que hacer polling busy-wait sobre el canal si no tiene nada más que hacer hasta que arriba un mensaje.

Asumimos que el acceso a los contenidos de cada canal es indivisible y que el reparto de mensajes es confiable y libre de error. Así, cada mensaje que es enviado a un canal es eventualmente repartido, y los mensajes no se corrompen. Dado que cada canal es también una cola FIFO, los mensajes serán recibidos en el orden en el cual fueron agregados al canal.

Como ejemplo veamos un proceso que recibe un stream de caracteres desde un canal *input*, ensambla los caracteres en líneas y envía las líneas resultantes a un segundo canal *output*. El carácter de retorno de carro, CR, indica el final de una línea, que puede ser a lo sumo de MAXLINE caracteres. Este proceso es un ejemplo de filtro pues transforma un stream de caracteres en un stream de líneas:

```
chan input(char), output([1:MAXLINE] char)
Char_to_Line:: var line[1:MAXLINE] : char, i : int := 1
        do true →
            receive input(line[i])
            do line[i] ≠ CR and i < MAXLINE →
                i := i + 1; receive input(line[i])
            od
            send output(line); i := 1
        od
```

Los canales serán declarados globales a los procesos, como en el ejemplo, pues son compartidos por procesos. Cualquier proceso puede enviar o recibir por un canal. Cuando los canales son usados de esta manera son llamados *mailboxes*. Sin embargo, en varios ejemplos que consideraremos, cada canal tendrá exactamente un receptor, aunque puede tener muchos emisores. En este caso, un canal es llamado *input port* pues provee una ventana en el proceso receptor. Si un canal tiene solo un emisor y un receptor, es llamado *link* pues provee un path directo del proceso emisor al receptor.

Usualmente un proceso querrá demorarse cuando ejecuta **receive**, pero no siempre. Por ejemplo, el proceso podría tener otro trabajo útil que hacer si un mensaje aún no está disponible para recepción. O, un proceso tal como un scheduler puede necesitar examinar todos los mensajes encolados para seleccionar el mejor para ser el próximo en ser servido. Para determinar si la cola de un canal está vacía, un proceso puede llamar a la función booleana:

empty(*ch*)

Esta función es true si el canal *ch* no contiene mensajes. A diferencia de la primitiva correspondiente para monitores, si un proceso llama a **empty** y retorna true, de hecho pueden encolarse mensajes cuando el proceso continúa su ejecución. Más aún, si un proceso llama a **empty** y retorna false, puede no haber ningún mensaje encolado cuando trata de recibir uno (esto no puede ocurrir si el proceso es el único que recibe por ese canal). En resumen, **empty** es una primitiva útil, pero se debe tener cuidado al usarla.

Ejemplo: Consideremos un programa en el cual *P1* envía un valor *x* a *P2*, el cual lo incrementa y lo devuelve a *P1*:

```
chan ch1(int), ch2(int)
P1:: var x : int := 0
     send ch2(x)
     R1: receive(ch1(x))
P2:: var y : int
     R2: receive ch2(y)
     send ch1(y+1)
```

FILTROS: UNA RED DE ORDENACION

La clave para entender los programas basados en mensajes es entender las suposiciones de comunicación. Por lo tanto, la clave para desarrollar un proceso que emplea MP es primero especificar las suposiciones de comunicación. Dado que la salida de un proceso filtro es una función de su entrada, la especificación apropiada es una que relaciona el valor de los mensajes enviados a los canales de salida con los valores de los mensajes recibidos por los canales de entrada. Las acciones que el filtro toma en respuesta a la entrada recibida debe asegurar esta relación cada vez que el filtro envía una salida.

Para ilustrar cómo son derivados y programados los filtros, consideremos el problema de ordenar una lista de *n* números en orden ascendente. La manera más directa de resolver el problema es escribir un único proceso filtro, *Sort*, que recibe la entrada desde un canal, emplea uno de los algoritmos de ordenación standard, y luego escribe el resultado en otro canal. Sean *input* el canal de entrada y *output* el de salida. Asumimos que los *n* valores a ordenar son enviados a *input* por algún proceso no especificado. Entonces el objetivo del proceso ordenador es asegurar que los valores enviados a *output* están ordenados y son una permutación de los valores recibidos. Sea que *sent*[i] indica el i-ésimo valor enviado a *output*. Entonces el objetivo es especificado por el siguiente predicado:

SORT: ($\forall$i: 1 $\leq$ i < n: sent[i] $\leq$ sent[i+1] $\wedge$ los valores enviados a
output son una permutación de los valores recibidos de input

Un outline del proceso *Sort* es:

```
Sort:: recibir todos los número desde el canal input
       ordenar los números
       enviar los números ordenados al canal output
```

Dado que **receive** es una primitiva bloqueante, algo práctico para *Sort* es determinar cuándo ha recibido todos los números. Una solución es que *Sort* conozca el valor de *n* de antemano. Una solución más general es que *n* sea el primer valor. Una solución más general aún es finalizar el stream de entrada con un valor *centinela*, el cual es un valor especial que indica que todos los número fueron recibidos. Esta solución es la más general pues el proceso que produce la entrada no necesita conocer en principio cuántos valores enviará a *input*. Asumiendo que los finales de *input* y *output* son marcados con centinelas, entonces el objetivo, SORT, es modificado levemente reemplazando *n* por *n*+1 para contar los dos centinelas.

Si los procesos son objetos “heavyweight” (como en la mayoría de los SO), la anterior aproximación con frecuencia sería la manera más eficiente de resolver el problema de ordenación. Sin embargo, una aproximación distinta (más directa de implementar en hardware) es emplear una red de pequeños procesos que ejecutan en paralelo e interactúan para resolver el problema. Hay muchas clases de redes de ordenación; aquí se presenta una *merge network*.

La idea es mezclar repetidamente (y en paralelo) dos listas ordenados en una lista ordenada más grande. La red es construida a partir de filtros *Merge*. Cada proceso *Merge* recibe valores desde dos streams de entrada ordenados, *in1* e *in2*, y produce un stream de salida ordenado, *out*. Asumimos que los finales de los streams de entrada son marcados por un centinela EOS. También que *Merge* agrega EOS al final del stream de salida. Si hay *n* valores de entrada, sin contar el centinela, entonces cuando *Merge* termina lo siguiente es true:

MERGE: in1 e in2 están vacíos $\wedge$ sent[n+1] = EOS $\wedge$
($\forall$i: 1 $\leq$ i < n: sent[i] $\leq$ sent[i+1] $\wedge$
los valores enviados a out son una permutación de
los valores recibidos de in1 e in2

La primera línea de MERGE dice que todas las entradas fueron consumidas y EOS fue agregado al final de *out*; el segunda línea dice que la salida está ordenada; la tercera línea dice que todos los datos de entrada fueron sacados.

Una manera de implementar *Merge* es recibir todos los valores de entrada, mezclarlos, y luego enviar la lista mezclada a *out*. Pero esto requiere almacenar todos los valores de entrada. Dado que los streams de entrada están ordenados, una mejor manera para implementar *Merge* es comparar repetidamente los próximos dos valores recibidos desde *in1* e *in2* y enviar el menor a *out*. Sean *v1* y *v2* estos valores. Esto sugiere el siguiente outline de proceso:

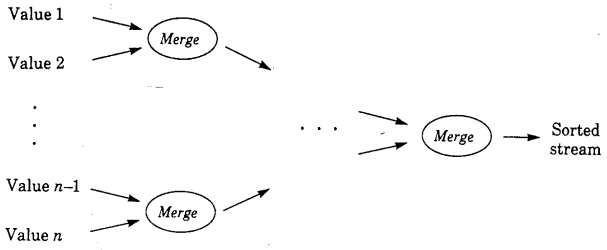
```
char in1(int), in2(int), out(int)
Merge:: var v1, v2 : int
        receive in1(v1); receive in2(v2)
        do más entradas al proceso →
              send menor valor a out
              receive otro valor desde in1 o in2
        od
        send out(EOS)     {MERGE}
```

Como caso especial, después de que *Merge* recibió todos los valores desde un stream de entrada, simplemente puede agregar los valores desde el otro stream de entrada a *out*.

Podemos tener un loop invariant apropiado por las técnicas de reemplazar una constante por una variable y borrar un conjuntor. En particular, borramos la primera línea de MERGE y reemplazamos la constante *n* en la segunda línea por la variable auxiliar *size*[*sent*[*out*] - 1]. Expandiendo el loop y manejando los casos especiales, tenemos el siguiente programa final:

```
char in1(int), in2(int), out(int)
Merge:: var v1, v2 : int
        receive in1(v1); receive in2(v2)
        do v1 ≠ EOS and v2 ≠ EOS →
            if v1 ≤ v2 → send out(v1); receive in1(v1)
            □ v2 ≤ v1 → send out(v2); receive in2(v2)
            fi
        □ v1 ≠ EOS and v2 = EOS → send out(v1); receive in1(v1)
        □ v1 = EOS and v2 ≠ EOS → send out(v2); receive in2(v2)
        od
        send out(EOS)     {MERGE}
```

Para formar una red de ordenación, empleamos una colección de procesos *Merge* y arreglos de canales de entrada y salida. Asumiendo que el número *n* de valores de entrada es potencia de 2, los procesos y canales son conectados de manera que el patrón de comunicación resultante forma un árbol:



La información en la red de ordenación fluye de izquierda a derecha. A cada nodo de la izquierda se le dan dos valores de entrada, los cuales mezcla para formar un stream de dos valores ordenados. Los siguientes nodos forman streams de 4 valores ordenados, y así siguiendo. El nodo de más a la derecha produce el stream ordenado final. La red contiene *n*-1 procesos; el ancho de la red es $\log_2 n$.

Para realizar esta red, los canales de entrada y salida necesitan ser compartidos. En particular, el canal de salida usado por una instancia de *Merge* necesita ser el mismo que uno de los canales de entrada usada por la siguiente instancia de *Merge* en el grafo. Esto puede ser programado de dos maneras. La primera es usar *static naming*: declarar todos los canales como un arreglo global, y que cada instancia de *Merge* reciba de dos elementos del arreglo y envíe a otro elemento. Esto requiere embeber el árbol en el arreglo para que los canales accedidos por *Merge* sean una función de *i*. La segunda forma es usar *dynamic naming*: declarar todos los canales globales, parametrizar los procesos y darle a cada proceso tres canales cuando es creado. Esto hace más fácil la programación de los procesos *Merge* pues cada uno es textualmente idéntico. Sin embargo, requiere tener un proceso principal que dinámicamente cree y pase canales a los procesos *Merge*.

Un atributo clave de los filtros como *Merge* es que podemos interconectarlos de distintas maneras. Todo lo que se necesita es que la salida producida por un filtro cumpla las suposiciones de entrada del otro. Una consecuencia importante de este atributo es que podemos reemplazar un proceso filtro (o una red de filtros) por una red diferente. Por ejemplo, podemos reemplazar el proceso *Sort* descripto antes por una red de procesos *Merge* más un proceso (o red) que distribuye los valores de entrada a la red de merge.

Las redes de filtros pueden usarse para resolver una variedad de problemas de programación paralela.

CLIENTES Y SERVIDORES

Recordemos que un servidor es un proceso que repetidamente maneja pedidos de procesos clientes. Mostraremos cómo programar clientes y servidores. El primer ejemplo muestra cómo transformar monitores en servidores y cómo implementar manejadores de recursos usando MP. También muestra la dualidad entre monitores y MP: cada uno puede simular directamente al otro.

Luego mostramos cómo implementar un disk driver self-scheduling y un file server. El disk driver ilustra una tercera manera de estructurar una solución al problema introducido en el cap. 6. El file server ilustra una importante técnica de programación llamada *continuidad conversacional*. Ambos ejemplos también ilustran estructuras de programa que son soportadas directamente por MP, en el sentido de que dan soluciones más compactas que las de los anteriores mecanismos.

Monitores Activos

Recordemos que un monitor es un manejador de recurso. Encapsula variables permanentes que registran el estado del recurso, y provee un conjunto de procedures que son llamados para acceder al recurso. Los procedures ejecutan con exclusión mutua y usan variables condición para sincronización por condición. Aquí mostramos cómo simular estos atributos usando procesos server y MP. En síntesis: mostramos cómo programar monitores como procesos activos en lugar de como colecciones pasivas de procedures.

Asumimos por el momento que un monitor provee solo una operación *op* y que no usa variables condición. Entonces la estructura del monitor es:

```
monitor Mname     # Invariante M1
  var variables permanentes
  código de inicialización
  procedure op(formales) cuerpo de op end
end
```

Para simular *Mname* usando MP, empleamos un proceso server *Sname*. Las variables permanentes de *Mname* se convierten en variables locales de *Sname* (que es un cuidador de esas variables). *Sname* primero inicializa las variables y luego ejecuta un loop permanente en el cual repetidamente sirve "llamados" de *op*. Un llamado se simula teniendo un proceso cliente que envía un mensaje a un canal de request, y luego recibe el resultado desde un canal de reply. El server repetidamente recibe desde el canal de request y envía resultados a canales de reply. Los parámetros formales en *Mname* se convierten en variables adicionales locales a *Sname*. Para evitar que un cliente vea el resultado que es para otro, cada cliente necesita su propio canal de resultado privado. Si son declarados como un arreglo global, un cliente necesita pasar el índice de su elemento privado del arreglo de resultado al server como parte del mensaje de request. (Algunas notaciones de MP permite al proceso receptor determinar la identidad del emisor).

El outline del server y los clientes es:

```
chan request(int, tipos de valores formales)
chan reply[1:n](tipos de resultados formales)
Sname:: var variables permanentes
        var index : int, valores formales, resultados formales
        código de inicialización
        do true → { loop invariant M1 }
                  receive request(index, valores formales)
                  cuerpo de op
                  send reply[index](resultados formales)
        od
Client[i:1..n]:: send request(i, argumentos valor)       # "llama" a op
                 receive reply[i](argumentos resultado)  # espera el reply
                 .....
```

El invariante *M1* se convierte en el loop invariant del proceso *Sname*. Es true después de ejecutar el código de inicialización y antes y después de que es servido cada request.

Este programa emplea static naming pues los canales son globales a los procesos y son referenciados directamente. En consecuencia, cada proceso debe ser codificado cuidadosamente para usar los canales correctos. Por ejemplo, *Client*[i] no debe usar el canal de reply de otro *Client*[j]. Alternativamente, emplearíamos dynamic naming teniendo que cada cliente cree un canal de reply privado, el cual le pasa a *Sname* como primer campo de *request* en lugar del índice entero. Esto aseguraría que los clientes no podrían acceder los canales de otros. También permitiría que el número de clientes varíe dinámicamente.

En general, un monitor tiene múltiples procedures. Para extender el programa anterior a manejar este caso, un cliente necesita indicar qué operación está llamando. Esto se hace incluyendo un argumento adicional en los mensajes de request; el tipo de este argumento es un tipo enumerativo con un literal para cada clase de operación. Dado que las distintas operaciones pueden tener distintos valores y resultados formales, necesitamos distinguir entre ellos. Esto puede hacerse usando un registro variante o un tipo unión.

El outline para clientes y un server con múltiples operaciones es el que sigue. La sentencia **if** en el server es como un case, con un branch para cada clase de operación. El cuerpo de cada operación recupera los argumentos desde *args* y pone los valores resultado en *results*. Después de que termina el **if**, *Sname* envía estos resultados al cliente apropiado:

```
type op_kind = enum(op1, ..., opn)
type arg_type = union(arg1 : atype1, ..., argn : atypen )
```

```
type result_type = union(res1 : rtype1, ..., resn : rtypen )
chan request(int, op_kind, arg_type)
chan reply[1:n](res_type)
Sname:: var variables permanentes
        var index : int, kind : op_kind, args : arg_type, results : res_type
        código de inicialización
        do true → { loop invariant M1 }
                  receive request(index, kind, args)
                  if kind = op1 → cuerpo de op1
                  □ ...
                  □ kind = opn → cuerpo de opn
                  fi
                  send reply[index](results)
        od
Client[i:1..n]:: var myargs : arg_type, myresults : result_type
                 poner los valores de los argumentos en myargs
                 send request(i, opi, myargs)      # "llama" a opi
                 receive reply[i](myresults)       # espera el reply
                 .....
```

Hemos asumido que *Mname* no emplea variables condición. Por lo tanto, *Sname* nunca necesita demorarse mientras sirve un pedido pues el cuerpo de cada operación será una secuencia de sentencias secuenciales. Ahora mostramos cómo manejar el caso general de un monitor con múltiples operaciones y sincronización por condición interna. (Los clientes no cambian pues solo llaman a una operación y esperan respuesta).

Para ver cómo trasladar un monitor con variables condición a un proceso server, empezamos considerando un ejemplo específico y luego describimos cómo generalizar el ejemplo. En particular, consideremos el problema de manejar un recurso de múltiple unidad (como memoria o bloques de archivo). Los clientes adquieren unidades del recurso, las usan, y luego las liberan al manejador. Por simplicidad, los clientes adquieren y liberan unidades de a una a la vez. El monitor de este manejador de recurso es el siguiente (usamos el método de passing the condition pues esta estructura de programa es la más fácil de trasladar a un proceso server. Las unidades libres se almacenan en un conjunto, el cual es accedido por operaciones *insert* y *remove*):

```
monitor Resource_Allocator
  var avail : int := MAXUNITS, units : set of int := valores iniciales
  var free : cond    # signal cuando se libera una unidad y free no está vacío
  procedure acquire( res id : int )
    if avail = 0 → wait(free)
    □ avail > 0 → avail := avail - 1
    fi
    id := remove(units)
  end
  procedure release( id : int )
    insert(id,units)
    if empty(free) → avail := avail + 1
    □ not empty(free) → signal(free)
    fi
  end
end
```

El monitor tiene dos operaciones, de modo que el proceso server equivalente tendrá la estructura general descripta. Una diferencia clave es que, si no hay unidades disponibles, el proceso server no puede esperar cuando está sirviendo un pedido. Debe salvar el pedido y diferir el envío de un reply. Cuando una unidad es liberada, el server necesita atender un pedido salvado, si hay, enviando la unidad liberada al que la solicitó.

El outline para el server de alocación de recurso y sus clientes es el siguiente. El server ahora tiene dos **if** anidados. El exterior tiene branches para cada clase de operación, y el interno corresponde a las sentencias **if** en los procedures del monitor. Después de enviar un mensaje de request, un cliente espera a recibir una unidad. Sin embargo, después de enviar un mensaje de release, el cliente no espera a que el mensaje sea procesado pues no hay necesidad de hacerlo:

```
type op_kind = enum(ACQUIRE,RELEASE)
chan request(int, op_kind, unitid : int)
chan reply[1:n](int)
Allocator:: var avail : int := MAXUNITS, units : set of int
            var pending : queue of int
            var index : int, kind : op_kind, unitid : int
            código para inicializar units a los valores apropiados
            do true →
               receive request(index, kind, unitid)
               if kind = ACQUIRE →
                    if avail > 0 →     # atiende el pedido ahora
                         avail := avail - 1; unitid := remove(units)
                         send reply[index](unitid)
                    □ avail = 0 →      # recuerda el pedido
                         insert(pending, index)
                    fi
               □ kind = RELEASE →
                    if empty(pending) →      # retorna unitid
                         avail := avail + 1; insert(units,unitid)
                    □ not empty(pending) →     # aloca unitid
                         index := remove(pending)
                         send reply[index](unitid)
                    fi
               fi
            od
Client[i:1..n]:: var unitid : int
            send request(i, ACQUIRE, 0)        # “llama” a request
            receive reply[i](unitid)
            # usa el recurso unitid, y luego lo libera
            send release(i, RELEASE, unitid)
            .....
```

Este ejemplo ilustra cómo simular un monitor específico con un proceso server. Podemos usar el mismo patrón para simular cualquier monitor que sea programado usando la técnica de passing the condition. Sin embargo, varios de los monitores vistos en el cap. 6 tienen **wait** embebidos en loops o sentencias **signal** ejecutadas incondicionalmente. Para simular tales sentencias **wait**, el server necesitaría salvar los pedidos pendientes y también registrar qué acciones debería tomar cuando el pedido puede ser servido. Para simular un **signal** incondicional, el server necesita chequear la cola de pedidos pendientes. Si está vacía, el server no hace nada; si hay un pedido pendiente, el server remueve uno de la cola y lo procesa *después* de procesar la operación que contiene el **signal**.

El monitor *Resource_Allocator* y el server *Allocator* muestran la dualidad entre monitores y MP: hay una correspondencia directa entre los mecanismos de los monitores y los de MP.

Dado que los cuerpos de los procedures del monitor tienen duales directos en los brazos de la sentencia case del server, la performance relativa de los programas basados en monitor versus en mensajes depende solo de la eficiencia relativa de la implementación de los distintos mecanismos. En máquinas de memoria compartida, los llamados a procedure y las acciones sobre variables condición tienden a ser más eficientes que las primitivas de MP. Por esta razón, la mayoría de los SO para tales máquinas se basan en una implementación estilo monitor. Por otro lado, la mayoría de los sistemas distribuidos se basan en MP pues es más eficiente y la abstracción apropiada. También es posible combinar aspectos de ambos estilos e implementaciones (cap. 9) como en RPC y rendezvous.

Un server de Self-Scheduling de disco

En el cap. 6 consideramos el problema de hacer scheduling de acceso a una cabeza de disco movible, y dimos dos estructuras de solución distintas. En la primera, el scheduler de disco era un monitor separado del disco. Así los clientes primero llamaban al scheduler para pedir el acceso, luego usaban el disco, y finalmente llamaban al scheduler para liberar el acceso. En la segunda, el scheduler era un intermediario entre los clientes y un proceso servidor del disco. Así los clientes tenían que llamar a una sola operación del monitor.

Rápidamente podríamos pasar ambas estructuras a programas basados en mensajes implementando el monitor como un server que usa las técnicas de la sección previa. Sin embargo, con MP, es posible una estructura más simple. En particular, podemos combinar el intermediario y el disk driver en un único proceso server self-scheduling. En todos los casos asumimos que el disco es controlado por un proceso server que realiza todos los accesos a disco. Las diferencias principales entre las tres estructuras son la interfase del cliente y el número de mensajes que deben ser intercambiados por acceso a disco.

Cuando el scheduler está separado del disk server, deben intercambiarse 5 mensajes por acceso a disco: dos para pedir scheduling y tener un reply, dos para pedir el acceso a disco y tener un reply, y uno para liberar el disco. Un cliente está involucrado en las 5 comunicaciones. Cuando el scheduler es un intermediario, deben intercambiarse 4 clases de mensajes. El cliente tiene que enviar un pedido y esperar recibir un reply, y el disk driver tiene que solicitarle al scheduler el próximo pedido y tomar el reply. Un driver de disco self-scheduling tiene la estructura más atractiva. En particular, se necesitan intercambiar sólo dos mensajes.

Si el proceso disk driver no hiciera scheduling (es decir, fuera FCFS) entonces tendría la estructura del server *Sname*. Para hacer scheduling, el driver debe examinar todos los pedidos pendientes, lo cual significa que debe recibir todos los mensajes que están esperando en el canal *request*. Esto lo hace ejecutando un loop que termina cuando el canal *request* está vacío y hay al menos un pedido salvado. Luego el driver selecciona el mejor pedido, accede al disco, y finalmente envía una respuesta al cliente que envió el pedido. El driver puede usar cualquiera de las políticas de scheduling ya descriptas.

A continuación se da un outline del disk driver que emplea la política de scheduling SST (shortest seek time). El driver almacena los pedidos pendientes en una de dos colas ordenadas, *left* o *right*, dependiendo de si el pedido está a la izquierda o la derecha de la posición corriente de la cabeza del disco. Los pedidos en *left* son ordenados por valor decreciente de cilindro; los de *right* son ordenados incrementando el valor del cilindro. El invariante para el loop externo del proceso driver es:

SST: *left* es una cola ordenada del *cyl* más grande al más chico $\wedge$
todos los valores de *cyl* en *left* son $\leq$ *headpos* $\wedge$
right es una cola ordenada del *cyl* más chico al más grande $\wedge$
todos los valores de *cyl* en *right* son $\geq$ *headpos* $\wedge$
(*nsaved* = 0) $\Rightarrow$ *left* y *right* están vacíos

```
chan request(index, cylinder : int; otros tipos de argumentos)
  # otros argumentos indican lectura o escritura, bloque, buffer, etc
chan reply[1:n](tipos resultado)
Disk_Driver:: var left, right : ordered queue of (index : int, cyl : int, ...)
              var headpos : int := 1, nsaved := 0
              var index : int, cyl : int, args : otros tipos de argumentos
              do true →      { SST }
                 do not empty(request) or nsaved = 0 →
                    # espera el primer pedido o recibe otro
                    receive request(index,cyl,args)
                    if cyl ≤ headpos := insert(left, (index,cyl,args))
                    □ cyl ≥ headpos := insert(right, (index,cyl,args))
                    fi
                    nsaved := nsaved + 1
                 od
```

```
            # selecciona el mejor pedido salvado de left o right
            if size(left) = 0 → (index,cyl,args) := remove(right)
            □ size(right) = 0 → (index,cyl,args) := remove(left)
            □ size(right) > 0 and size(left) > 0 →
                  remove un elemento de left o right dependiendo de
                  qué valor salvado de cyl está más cerca de headpos
            fi
            headpos := cyl; nsaved := nsaved - 1
            accede al disco
            send reply[index](results)
      od
```

Se usa la primitiva **empty** en la guarda del loop interno para determinar si hay más mensajes encolados en el canal *request*. Este es un ejemplo de la técnica de programación llamada *polling*. En este caso, el proceso disk driver repetidamente polls el canal *request* para determinar si hay pedidos pendientes. Si hay, el driver recibe otro, de modo que tiene más pedidos de donde elegir. Si no hay (en el momento que **empty** es evaluada) el driver sirve el mejor pedido pendiente. El polling también es útil en otras situaciones y con frecuencia se usa dentro del hardware (por ejemplo, para arbitrar el acceso a un bus de comunicación).

File Servers: Continuidad Conversacional

Como ejemplo final de la interacción cliente/servidor, presentamos una manera de implementar file servers, que son procesos que proveen acceso a archivos en almacenamiento secundario (por ej, archivos en disco). Para acceder a un archivo, un cliente primero abre el archivo. Si la apertura es exitosa (el archivo existe y el cliente tiene permiso para acceder a él) entonces el cliente hace una serie de pedidos de read y write. Eventualmente el cliente cierra el archivo.

Supongamos que puede haber hasta *n* archivos abiertos a la vez y que el acceso a cada archivo abierto es provisto por un proceso file server separado. Por lo tanto, hay *n* de tales procesos. Para abrir un archivo, un cliente necesita adquirir un file server que esté libre para interactuar con él. Si todos los file servers son idénticos, cualquiera que esté libre sirve.

Podríamos alocar FS a clientes usando un proceso alocador separado. Sin embargo, dado que todos son idénticos y los canales de comunicación son compartidos, hay una forma mucho más simple. En particular, sea *open* un canal global. Para adquirir un FS, un cliente envía un pedido a *open*. Cuando están ociosos, los FS tratan de recibir de *open*. Un pedido específico en *open* de un cliente será recibido por uno de los FS ociosos. Ese FS envía un reply al cliente, luego espera los pedidos de acceso. Un cliente envía estos a un canal diferente, *access*[*i*], donde *i* es el índice del FS que se alocó al cliente. Así, *access* es un arreglo de *n* canales. Eventualmente el cliente cierra el archivo, y en ese momento el FS se vuelve ocioso y espera un nuevo pedido de apertura.

A continuación se dan outlines para los FS y sus clientes. Un cliente envía sus pedidos de acceso al archivo (READ y WRITE) al mismo canal server. Esto es necesario pues el FS no puede en general saber el orden en el cual los pedidos son hechos y por lo tanto no puede usar distintos canales para cada uno. Por esta razón, cuando un cliente quiere cerrar un canal, envía un pedido de CLOSE al mismo canal de acceso:

```
type kind = enum(READ,WRITE,CLOSE)
chan open(fname : string[*], clientid : int)
chan access[1:n](kind : int, otros tipos)   # buffer, número de bytes, etc
chan open_reply[1:m](int)  #el campo es índice de server o indicación de error
chan access_reply[1:m](tipos resultado)     # datos, flags de error, etc
File_Server[i:1..n]:: var fname : string[*], clientid : int
                      var k : kind, args : otros tipos de argumento
                      var more : bool := false
                      var buffer local, cache, disk address, etc
                      do true →
                         receive open(fname,clientid)
                         # abre archivo fname; si tiene éxito entonces:
```

```
                    send open_reply[clientid](i); more := true
                   do more →
                      receive access[i](k,args)
                      if k = READ → procesa pedido de lectura
                      □ k = WRITE → procesa pedido de escritura
                      □ k = CLOSE → cierra archivo; more := false
                      fi
                   od
                od
Client[j:1..m]:: send open("xxx", j)          # abre archivo "xxx"
                 receive open_reply[j](serverid)    # toma la id del server
                 # usa el archivo y luego lo cierra ejecutando lo siguiente
                 send access[serverid](argumentos de acceso)
                 receive access_reply[j](resultados)
                 .....
```

La interacción entre un cliente y un server es un ejemplo de *continuidad conversacional*. En particular, un cliente comienza una "conversación" con un FS cuando ese server recibe el pedido de apertura. Luego el cliente sigue conversando con el mismo server. Esto se programa haciendo que el server primero reciba de *open*, luego repetidamente recibe de su elemento de *access*.

El programa ilustra una posible manera de implementar FS. Asume que *open* es un canal compartido por el que cualquier FS puede recibir un mensaje. Si cada canal puede tener solo un receptor, se necesita un proceso alocador de archivos separado. Ese proceso recibiría pedidos de apertura y alocaría un server libre a un cliente; los FS necesitarían decirle al alocador cuándo están libres.

La solución dada emplea un número fijo de *n* FS. En un lenguaje que soporte creación dinámica de procesos y canales, sería mejor crear FS y acceder canales dinámicamente. Esto es mejor pues en cualquier punto de tiempo habría solo tantos servers como los que se están usando; más aún, no habría un límite superior en el número de FS. En el otro extremo, podría haber simplemente un FS por disco. En este caso, o el FS o la interfase del cliente sería mucho más compleja. Esto es porque o el FS tiene que seguir la pista de la información asociada con todos los clientes que tienen archivos abiertos o los clientes tienen que pasar la información de estado de archivo con cada pedido.

ALGORITMOS HEARTBEAT

En un sistema jerárquico, los servers en niveles intermedios con frecuencia son también clientes de servers de más bajo nivel. Por ejemplo, el FS anterior podría procesar los pedidos de read y write comunicándose con un disk server como el que desarrollamos.

En esta sección y varias de las siguientes, examinamos una clase distinta de interacción server en la cual los servers al mismo nivel son peers que cooperan para proveer un servicio. Este tipo de interacción se da en computaciones distribuidas en la cual no hay un solo server que tiene toda la información necesaria para servir un pedido del cliente. También introducimos distintos paradigmas de programación que pueden ser usados para resolver problemas similares, y varias técnicas para desarrollar soluciones a problemas de programación distribuida.

En esta sección consideramos el problema de computar la topología de una red. Este problema es representativo de una gran clase de problemas de intercambio de información distribuida que se da en redes. Se supone una red de procesadores conectados por canales bidireccionales. Cada procesador se puede comunicar solo con sus vecinos y conoce solo los links con sus vecinos. El problema es para cada procesador determinar la topología de la red entera, es decir, el conjunto completo de links (por ej, para decidir el ruteo de mensajes).

Cada procesador es modelizado por un proceso, y los links de comunicación con canales compartidos. Resolvemos este problema asumiendo primero que todos los procesos tienen acceso a una memoria compartida, lo que no es realista para este problema. Luego refinamos la solución en una computación distribuida replicando variables globales y teniendo procesos vecinos interactuando para intercambiar su información local. En particular, cada proceso ejecuta una secuencia de iteraciones. En cada iteración, un proceso envía su conocimiento local de la topología a todos sus vecinos, luego recibe la información de ellos y la combina con la suya. La computación termina cuando todos los procesos aprendieron la topología de la red entera.

Llamamos a este tipo de interacción entre procesos *algoritmo heartbeat* pues las acciones de cada nodo son como el latido de un corazón: primero se expande, enviando información; luego se contrae, incorporando nueva información. El mismo tipo de algoritmo puede usarse para resolver otros problemas, especialmente los de computaciones iterativas paralelas.

Topología de red: Solución con Variables Compartidas

Especificación del problema: Son dados *n* nodos, uno por procesador. Cada nodo *p* puede comunicarse solo con sus vecinos, e inicialmente todo lo que conoce sobre la topología es su conjunto de vecinos. Asumimos que la relación de vecino es simétrica: para cada par de nodos *p* y *q*, *p* es vecino de *q* si y solo si *q* es vecino de *p*. El problema es computar el conjunto *top* de todos los links, es decir todos los pares (*p*,*q*) tales que *p* es vecino de *q*. Durante la computación, asumimos que la topología de la red es estática.

Representaremos cada nodo por un proceso *Node*[p:1..n]. Dentro de cada proceso, podemos representar los vecinos de un nodo por un vector booleano (bit) *links*[1:n] con el elemento *links*[q] true en *Node*[p] si *q* es vecino de *p*. Estos vectores se asumen inicializados con los valores apropiados. La topología final *top* puede ser representada por una matriz de adyacencia, con *top*[p,q] true si *p* y *q* son nodos vecinos. Específicamente, cuando la computación termina, el siguiente predicado debe ser verdadero ($links_p$ es el conjunto de vecinos del nodo *p*):

$$\text{TOPOLOGY: } (\ \forall\ p,q: 1 \le p \le n,\ 1 \le q \le n: top[p,q] \Leftrightarrow links_p[q]\ )$$

Para resolver el problema, asumimos por ahora que *top* es global a todos los procesos *Node* y que inicialmente todos los elementos de *top* son falsos. Entonces todo lo que cada proceso necesita hacer es almacenar el valor de su conjunto de vecinos en la fila apropiada de *top*. De hecho, dado que cada proceso asigna a una fila distinta de *top*, las asignaciones pueden ejecutar en paralelo sin interferencia. Esto lleva a la siguiente solución de variables compartidas (cuando cada proceso *Node*[p] termina, *top*[p,q] es true si *q* es un vecino de *p*, y falso en otro caso. El estado final del programa es la unión del estado final de cada proceso, con lo cual TOPOLOGY será true):

```
var top[1:n,1:n] : bool := ([n*n] false)
Node[p:1..n]:: var links[1:n] : bool
                    # inicialmente links[q] es true si q es un vecino de Node[p]
               fa q := 1 to n st links[q] → top[p,q] := true af
               { top[p,1:n] = links[1:n] }
```

Topología de red: Solución Distribuida

Una manera de convertir la solución anterior en un programa distribuido que emplea MP es tener un único proceso T que compute la topología. Esto se haría teniendo que cada nodo envía su valor de *links* a T, el cual copiaría el mensaje en la fila apropiada de *top*. Esto usa un número mínimo de mensajes (2n), pero tiene un problema: cómo hace un nodo para comunicarse con T si T no es uno de sus vecinos. Podríamos hacer que los nodos vecinos de T hagan forward de los mensajes de los otros nodos, pero el algoritmo sería muy asimétrico. Es preferible un algoritmo simétrico en el cual cada nodo ejecuta el mismo programa pues es más fácil de desarrollar y entender. Más aún, un algoritmo simétrico es más fácil de modificar para tratar con fallas potenciales de procesadores o links.

Para tener un algoritmo simétrico necesitamos una manera para que cada nodo compute por sí mismo la topología completa *top*. Inicialmente el nodo *p* conoce los links con sus vecinos. Si les pide a éstos los vecinos de ellos (enviando un mensaje y recibiendo un reply de cada uno) después de una ronda de mensajes, *p* conocerá la topología hasta dos links de sí mismo; es decir, conocerá sus links y los de sus vecinos. Si cada nodo hace lo mismo, luego de una ronda completa cada nodo conocerá la topología hasta dos links de sí mismo. Si cada nodo ejecuta otra ronda en la cual nuevamente intercambia lo que conoce con sus vecinos, después de dos rondas cada nodo conocerá la topología hasta tres links. En general, después de *r* rondas lo siguiente será true en cada nodo *p*:

ROUND: ($\forall$ q: 1 $\leq$ q $\leq$ n: (dist(p,q) $\leq$ r $\Rightarrow$ top[q,*] lleno)

Si cada nodo ejecuta un número adecuado de rondas, entonces a partir de ROUND podemos concluir que cada nodo computará la topología completa de la red. Asumimos por ahora que conocemos el *diámetro* D de la red (distancia entre el par de nodos más lejano). El siguiente programa da una solución (por simplicidad, las declaraciones de los canales de comunicación son para todos los procesos, aunque por supuesto para este problema asumimos que un nodo puede comunicarse solo con sus vecinos):

```
chan topology[1:n]([1:n,1:n] bool)      # un canal privado por nodo
Node[p:1..n]:: var links[1:n] : bool
                  # inicialmente links[q] true si q es vecino de Node[p]
               var top[1:n,1:n] : bool := ([n*n]false)   # links conocidos
               top[p,1..n] := links       # llena la fila para los vecinos
               var r : int := 0
               var newtop[1:n,1:n] : bool
               { top[p,1:n] = links[1:n] ∧ r = 0 }  { ROUND }
               do r < D →
                  # envía el conocimiento local de la topología a sus vecinos
                  fa q := 1 to n st links[q] → send topology[q](top) af
                  # recibe las topologías y hace or con su top
                  fa q := 1 to n st links[q] →
                        receive topology[p](newtop)
                        top := top or newtop
                  af
                  r := r + 1
               od
               { ROUND ∧ r = D }  { TOPOLOGY }
```

Como indicamos, ROUND es el loop invariant en cada nodo. Se hace true inicialmente seteando *top* con los vecinos del nodo e inicializando la variable local *r* a 0. Se preserva en cada iteración pues cada nodo intercambia toda su información local con sus vecinos en cada ronda. El **or** lógico se usa para unir una topología del vecino, recibida en *newtop*, con la topología local, almacenada en *top*. Cuando un nodo termina, ejecutó D rondas. Como por definición de D ningún nodo está más lejos que D links, *top* contiene la topología entera.

Hay dos problemas con este algoritmo. Primero, no sabemos el valor de D. Segundo, hay un excesivo intercambio de mensajes. Esto es porque los nodos cerca del centro de la red conocerán la topología entera tan pronto como hayan ejecutado las suficientes rondas para haber recibido la información de los nodos en los bordes de la red. En rondas subsecuentes, estos nodos no aprenderán nada nuevo, aunque seguirán intercambiando información con sus vecinos.

El loop invariant ROUND y las observaciones anteriores sugieren cómo solucionar ambos problemas. Después de *r* rondas, el nodo *p* conocerá la topología a distancia *r* de él. En particular, para cada nodo *q* dentro de la distancia *r* de *p*, los vecinos de *q* estarán almacenados en la fila *q* de *top*. Dado que la red es conectada, cada nodo tiene al menos un vecino. Así, el nodo *p* ejecutó las suficientes rondas para conocer la topología tan pronto como cada fila de *top* tiene algún valor true. En ese punto, *p* necesita ejecutar una última ronda en la cual intercambia la topología con sus vecinos; luego *p* puede terminar. Esta última ronda es necesaria pues *p* habrá recibido nueva información en la ronda previa. También evita dejar mensajes no procesados en los canales. Dado que un nodo podría terminar una ronda antes (o después) que un vecino, cada nodo también necesita decirle a sus vecinos cuando termina. Para evitar deadlock, en la última ronda un nodo debería intercambiar mensajes solo con sus vecinos que no terminaron en la ronda previa.

El siguiente programa da una solución para el problema de la topología de red:

```
chan topology[1:n](sender : int; done : bool; top : [1:n,1:n] bool)
Node[p:1..n]:: var links[1:n] : bool
                  # inicialmente links[q] true si q es vecino de Node[p]
               var active[1:n] : bool := links    # vecinos aún activos
               var top[1:n,1:n] : bool := ([n*n]false)   # links conocidos
               var r : int := 0, done : bool := false
               var sender : int, qdone : bool, newtop[1:n,1:n] : bool
               top[p,1..n] := links       # llena la fila para los vecinos
               { top[p,1:n] = links[1:n] ∧ r = 0  ∧  ¬done }
               { ROUND ∧ done ⇒ (todas las filas en top fueron llenadas}
               do not done  →
                  # envía conocimiento local de la topología a todos sus vecinos
                  fa q := 1 to n st links[q] → send topology[q](p,false,top) af
                  # recibe las topologías y hace or con su top
                  fa q := 1 to n st links[q] →
                          receive topology[p](sender,qdone,newtop)
                          top := top or newtop
                          if qdone → active[sender] := false fi
                  af
                  if todas las filas de top tiene un entry true → done := true fi
                  r := r + 1
               od
               { ROUND ∧ todas las filas en top fueron llenadas}  {TOPOLOGY}
               # envía topología a todos sus vecinos aún activos
               fa q := 1 to n st active[q] → send topology[q](p,done,top) af
               # recibe un mensaje de cada uno para limpiar el canal
               fa q := 1 to n st active[q] →
                   receive topology[p](sender,d,newtop)
               af
```

En este programa, *r* ahora es una variable auxiliar (se usa solo para facilitar la especificación del predicado ROUND). El loop invariant es ROUND y un predicado especificando que *done* es true solo si todas las filas de *top* fueron llenadas. Así, cuando el loop termina, un nodo escuchó de cada uno de los otros procesos, de modo que *top* contiene la topología de red completa.

El programa está libre de deadlock pues los **send**s se ejecutan antes que los **receive**s y un nodo recibe solo tantos mensajes en cada ronda como vecinos activos tiene. El loop termina en cada nodo pues la red está conectada y la información se propaga a cada vecino en cada ronda. La ronda final de **send**s y **receive**s asegura que cada vecino ve la topología y que no quedan mensajes sin recibir.

El algoritmo centralizado mencionado al inicio de esta sección requiere el intercambio de 2*n* mensajes, uno desde cada nodo al server central y uno de respuesta. El algoritmo descentralizado requiere el intercambio de más mensajes. El número real depende de la topología de la red pues el número de rondas que ejecuta un nodo depende de su distancia al nodo más lejano. Sin embargo, si *m* es el número máximo de vecinos que tiene cada nodo y *D* es el diámetro de la red, el número de mensajes que se intercambian está limitado por 2*n* * *m* * (*D* + 1). Esto es porque cada uno de los *n* nodos ejecuta a lo sumo *D* + 1 rondas durante las cuales intercambia 2 mensajes con cada uno de sus *m* vecinos. Si *m* y *D* son relativamente chicos comparados con *n*, entonces el número de mensajes no es mucho mayor que para el algoritmo centralizado. Además, estos pueden ser intercambiados en paralelo en muchos casos, mientras que un server centralizado espera secuencialmente recibir un mensaje de cada nodo antes de enviar cualquier respuesta.

El loop principal en cualquier algoritmo heartbeat tendrá la misma estructura básica: enviar mensajes a todos los vecinos, y luego recibir mensajes de ellos. Lo que contienen los mensajes y cómo son procesados depende de la aplicación. Por ejemplo, en una computación de grilla, los nodos intercambiarían valores de borde con sus vecinos, y la computación en cada ronda sería computar nuevos valores para puntos de grilla locales.

Otra diferencia entre instancias de algoritmos heartbeat es el criterio de terminación y cómo es chequeado. Para el problema de la topología, cada nodo puede determinar por sí mismo cuándo terminar. Esto es porque un nodo adquiere más información en cada ronda, y la información que ya tiene no cambia. En muchas computaciones en grilla, el criterio de terminación también se basa en información local (por ej, los valores de los puntos después de una ronda están dentro de epsilon de sus valores después de la ronda previa.

No siempre la terminación puede ser decidida localmente. Por ejemplo, consideremos usar una computación en grilla para hacer labelling de regiones en una imagen, con cada nodo en la red responsable de un bloque de la imagen. Dado que una región podría “serpentear” a lo largo de la imagen, un nodo no puede decir cuándo tiene toda la información de los nodos distantes; podría ver que no hay cambio en una ronda y obtener nueva información varias rondas más tarde. Tal computación puede terminar solo cuando no hay cambio en ningún lugar luego de una ronda. Así, los procesos necesitan comunicarse con un controlador central, intercambiar mensajes adicionales con los otros, o ejecutar el número de rondas del peor caso.

ALGORITMOS PROBE/ECHO

Los árboles y grafos se usan en muchos problemas de computación; por ejemplo, juegos, BD, y sistemas expertos. Son especialmente importantes en la computación distribuida pues la estructura de muchas computaciones distribuidas es un grafo en el cual los procesos son nodos y los canales de comunicación son aristas.

Depth-first search (DFS) es uno de los paradigmas de programación secuencial clásicos para visitar todos los nodos en un árbol o grafo. En un árbol, la estrategia DFS para cada nodo es visitar los hijos de ese nodo y luego retorna al padre. Esto es llamado DFS pues cada camino de búsqueda alcanza una hoja antes de que sea atravesado el próximo path; por ej, el camino en el árbol desde la raíz hasta la hoja más a la izquierda es atravesado primero. En un grafo general (el cual puede tener ciclos) se usa la misma aproximación, excepto que necesitamos marcar los nodos que son visitados para que las aristas sean atravesadas solo una vez.

Esta sección describe el paradigma probe/echo para computaciones distribuidas en grafos. Un probe es un mensaje enviado por un nodo a su sucesor; un echo es un reply subsecuente. Dado que los procesos ejecutan concurrentemente, los probes se envían en paralelo a todos los sucesores. El paradigma probe/echo es el análogo en programación concurrente a DFS, Primero ilustramos el paradigma probe mostrando cómo hacer broadcast de información a todos los nodos de una red. Luego agregamos el paradigma echo desarrollando un algoritmo distinto para construir la topología de una red.

Broadcast en una red

Asumimos que hay un nodo por procesador y que cada nodo puede comunicarse solo con sus vecinos. Supongamos que un nodo iniciador *i* quiere hacer broadcast de un mensaje (es decir, enviar alguna información a todos los otros nodos). Por ejemplo, *i* podría ser el sitio del coordinador de la red, que quiere hacer broadcast de información de nuevo estado a todos los otros sitios.

Si cada nodo es un vecino de *i* el broadcast sería trivial de implementar: el nodo *i* simplemente enviaría un mensaje directamente a cada otro nodo. Sin embargo, en la situación más realista en la cual solo un número de nodos son vecinos de *i*, necesitamos tener información forwared por los nodos hasta que estemos seguros que todos lo vieron. En síntesis, necesitamos enviar un probe a todos los nodos.

Asumimos que el nodo *i* tiene una copia local *top* de la topología completa de la red, computada, por ejemplo, como mostramos antes. Entonces una manera eficiente para que *i* haga broadcast de un mensaje es primero construir un *spanning tree* de la red, con él mismo como raíz del árbol. Un spanning tree de un grafo es un árbol cuyos nodos son todos los del grafo y cuyas aristas son un subconjunto de las del grafo, como muestra la figura:

Dado un spanning tree *T*, el nodo *i* puede hacer broadcast de un mensaje *m* enviando *m* junto con *T* a todos sus hijos en *T*. Luego de recibir el mensaje, cada nodo examina *T* para determinar sus hijos en el árbol, luego hace forward de *m* y *T* a todos ellos. El spanning tree es enviando junto con *m* pues los nodos que no son *i* no sabrían qué árbol usar.

El algoritmo completo se da a continuación. Dado que *T* es un spanning tree, eventualmente el mensaje alcanzará todo nodo; además, cada nodo lo recibirá exactamente una vez, de su padre en *T*. Un proceso separado, *Inititator*, en el nodo *i*, inicial el broadcast. Esto hace el algoritmo simétrico en cada nodo:

```
chan probe[1:n](span_tree : [1:n, 1:n] int, message_type)
Node[p:1..n]:: var span_tree[1:n,1:n] : int, m : message_type
               receive probe[p](span_tree,m)
               fa q := 1 to n st q es un hijo de p en span_tree →
                    send probe[q](span_tree,m)
               af
Initiator:: var i : int := índice del nodo que inicia el broadcast
          var top[1:n, 1:n] : int     # inicializado con la topología
          var T[1:n, 1:n] : int, m : message_type
          # computar el spanning tree de top y almacenarlo en T
          m := mensaje a emitir
          send probe[i](T, m)
```

Este algoritmo asume que el nodo iniciador conoce la topología completa, la cual usa para computar un spanning tree que guía el broadcast. Supongamos que cada nodo conoce solo sus vecinos. Todavía podemos broadcast un mensaje *m* a todos los nodos como sigue. Primero, el nodo *i* envía *m* a todos sus vecinos. Luego de recibir *m*, un nodo lo envía a todos sus otros vecinos. Si los links definidos por el conjunto de vecinos forman un árbol con raíz *i*, el efecto de esta aproximación es el mismo que antes. Sin embargo, en general, la red contendrá ciclos. Así, algún nodo podría recibir *m* desde dos o más vecinos. De hecho, dos vecinos podrían enviar el mensaje uno al otro al mismo tiempo.

Parecería que lo único que necesitamos hacer en el caso general es ignorar las múltiples copias de *m* que podría recibir un nodo. Sin embargo, esto lleva al siguiente problema. Después de recibir *m* la primera vez y enviarlo, un nodo no puede saber cuántas veces esperar para recibir *m* desde un nodo distinto. Si el nodo no espera nada, podrían quedar mensajes extra buffereados en alguno de los canales *probe*. Si un nodo espera un número fijo de veces, podría quedar en deadlock a menos que varios mensajes sean enviados; aún así, podría haber más.

La solución al problema de los mensajes no procesados es tener un algoritmo totalmente simétrico. En particular, dejamos que cada nodo que recibe *m* la primera vez envíe *m* a todos sus vecinos, incluyendo al que le envió *m*. Luego el nodo recibe copias redundantes de *m* desde todos sus otros vecinos; estos los ignora. El algoritmo es el siguiente:

```
chan probe[1:n](message_type)
Node[p:1..n]:: var links[1:n] : bool := vecinos del nodo p
               var num : int := número de vecinos
               var m : message_type
               receive probe[p](m)
               # envía m a todos los vecinos
               fa q := 1 to n st links[q] → send probe[q](m)  af
               # recibe num-1 copias redundantes de m
               fa q := 1 to num-1 → receive probe[q](m)  af
Initiator:: var i : int := índice del nodo que inicia el broadcast
          var m : message_type
          m := mensaje a emitir
          send probe[i](m)
```

El algoritmo broadcast usando un spanning tree causa que se envíen n-1 mensajes, uno por cada arista padre/hijo en el spanning tree. El algoritmo usando conjuntos de vecinos causa que dos mensajes sean enviados sobre cada link en la red, uno en cada dirección. El número exacto depende de la topología de la red, pero en general el número será mucho mayor que n-1. Por ejemplo, si la topología es un árbol con raíz en el proceso iniciador, se enviarán 2(n-1) mensajes. Sin embargo, el algoritmo del conjunto de vecinos no requiere que el iniciador conozca la topología y compute un spanning tree. En esencia, el spanning tree es construido dinámicamente; consta de los links por los que se envían las primeras copias de *m*. Además, los mensajes son más cortos en este algoritmo pues el spanning tree (n^2 bits) no necesita ser enviado en cada mensaje.

Ambos algoritmos asumen que la topología de la red no cambia. En particular, ninguno funciona correctamente si hay una falla de procesador o link de comunicación cuando el algoritmo está ejecutando. Si un nodo falla, no puede recibir el mensaje. Si un link falla, podría o no ser posible alcanzar los nodos conectados por el link.

Topología de red Revisitada

En secciones anteriores derivamos un algoritmo distribuido para computar la topología de una red comenzando con un algoritmo de memoria compartida y luego generando múltiples copias de los datos compartidos. Aquí resolvemos el mismo problema de manera completamente distinta. En particular, tenemos un nodo iniciador que junta los datos de topología local de cada uno de los otros nodos y luego disemina la topología completa a los otros nodos. La topología es reunida en dos fases. Primero, cada nodo envía un probe a sus vecinos. Luego, cada nodo envía un echo conteniendo la información de topología local al nodo del cual recibió el primer probe. Eventualmente, el nodo que inicia reunió todos los echoes. Entonces puede computar un spanning tree para la red y broadcast la topología completa usando alguno de los algoritmos ya vistos.

Asumimos por ahora que la topología de la red es acíclica; dado que es un grafo no dirigido, la estructura es un árbol. Sea el nodo *i* la raíz del árbol, y sea *i* el nodo iniciador. Así podemos reunir la topología como sigue. Primero *i* envía un probe a todos sus vecinos. Cuando estos nodos reciben un probe, lo envía a todos sus vecinos, y así siguiendo. Así, los probes se propagan a través del árbol. Eventualmente alcanzarán los nodos hoja. Dado que estos no tienen otros vecinos, comienzan la fase de echo. En particular, cada hoja envía un echo conteniendo su conjunto de vecinos a su padre en el árbol. Después de recibir echoes de todos sus hijos, un nodo los combina con su propio conjunto de vecinos y echoes la información a su padre. Eventualmente el nodo raíz recibirá echoes de todos sus hijos. La unión de estos contendrá la topología entera pues el probe inicial alcanzará cada nodo y cada echo contiene el conjunto de vecinos del nodo que hizo el echo junto con los de sus descendientes.

El algoritmo probe/echo completo para reunir la topología de una red en un árbol se muestra a continuación. La fase de probe es esencialmente el último algoritmo broadcast, excepto que no se hace broadcast de ningún mensaje; los mensajes probe solo indican la identidad del emisor. La fase de echo retorna la información de topología local hacia arriba en el árbol. En este caso, los algoritmos para los nodos no son completamente simétricos pues la instancia de *Node*[p] ejecutando en el nodo *i* necesita poder enviar su echo al *Initiator*. Después que *Initiator* recibe la topología final en *top*, puede computar un spanning tree y broadcast *top* a los otros nodos:

```
const source = i    # índice del nodo que inicia el algoritmo
chan probe[1:n](sender : int)
chan echo[1:n](links[1:n,1:n] : bool)   #parte de la topología
chan finalecho[1:n](links[1:n, 1:n] : bool)     # echo final a Initiator
Node[p:1..n]:: var links[1:n] : bool := vecinos del nodo p
                var localtop[1:n,1:n] : bool := ([n*n]false)
                localtop[p, 1:n] := links
                var newtop[1:n, 1:n] : bool
                var parent : int     # nodo desde quien se recibe el probe
               receive probe[p](parent)
               # envía probe a otros vecinos, que son hijos de p
               fa q := 1 to n st links[q] and q ≠ parent →
                     send probe[q](p)
```

```
                af
                # recibe echoes y los une en localtop
                fa q := 1 to n st links[q] and q ≠ parent →
                    receive echo[p](newtop)
                    localtop := localtop or newtop
                af
                if p = source → send finalecho(localtop)   # el nodo p es raíz
                □ p ≠ source → send echo[parent](localtop)
                fi
Initiator:: var top[1:n, 1:n] : bool
        send probe[source](source)   # comienza el probe en el nodo local
        receive finalecho(top)
```

Para computar la topología de una red que contiene ciclos, generalizamos el algoritmo anterior como sigue. Luego de recibir un probe, un nodo lo envía a todos sus otros vecinos, luego espera un echo de cada uno. Sin embargo, a causa de los ciclos y que los nodos ejecutan concurrentemente, dos vecinos podrían enviar cada uno otros probes casi al mismo tiempo. Los probes que no son el primero pueden ser echoed inmediatamente. En particular, si un nodo recibe un probe subsecuente mientras está esperando echos, inmediatamente envía un echo conteniendo una topología nula. Eventualmente un nodo recibirá un echo en respuesta a cada probe. En este punto, echoes la unión de su conjunto de vecinos y los echos que recibió.

El algoritmo probe/echo general para computar la topología de red es el siguiente. Dado que un nodo podría recibir probes subsecuentes mientras espera echoes, los dos tipos de mensajes tienen que ser mezclados en un canal:

```
const source = i    # índice del nodo que inicia el algoritmo
type kind = enum(PROBE,ECHO)
chan probe_echo[1:n](kind,sender : int, links[1:n, 1:n] int)
chan finalecho[1:n](links[1:n, 1:n] : int)     # echo final a Initiator
Node[p:1..n]:: var links[1:n] : bool := vecinos del nodo p
        var localtop[1:n,1:n] : bool := ([n*n]false)
        localtop[p, 1:n] := links
        var newtop[1:n, 1:n] : bool
        var first : int     # nodo desde quien se recibe el probe
        var k : kind, sender : int
        var need_echo : int := número de vecinos - 1
       receive probe_echo[p](k, sender, newtop)   # toma el probe
       first := sender
       # envía probe a todos los otros vecinos
       fa q := 1 to n st links[q] and q ≠ first →
            send probe_echo[q](PROBE,p,∅)
       af
       do need_echo > 0 →
            # recibe echoes o probes de sus vecinos
            receive probe_echo[p](k,sender,newtop)
            if k = PROBE →
                 send probe_echo[sender](ECHO,p, ∅)
            □ k = ECHO →
                 localtop := localtop or newtop
                 need_echo := need_echo - 1
            fi
       od
       if p = source → send finalecho(localtop)
       □ p ≠ source → send probe_echo[first](ECHO,p,localtop)
       fi
Initiator:: var top[1:n, 1:n] : bool      # topología de la red
        send probe_echo[source](PROBE,source,∅)
        receive finalecho(top)
```

Este algoritmo probe/echo para computar la topología de una red requiere menos mensajes que el algoritmo heartbeat. Se envían dos mensajes a lo largo de cada link que es una arista del spanning tree de los primeros probes (uno para el probe y otro para el echo). Otros links llevan 4 mensajes (un probe y un echo en cada dirección). Para diseminar la topología desde el *Initiator* a todos los nodos usando el algoritmo broadcast requeriríamos otros *n* mensajes. En cualquier evento, el número de mensajes es proporcional al número de links. Para computaciones que diseminan o reúnen información en grafos, los algoritmos probe/echo son más eficientes que los heartbeat. En contraste, los algoritmos heartbeat son apropiados y necesarios para muchos algoritmos iterativos paralelos en los cuales los nodos repetidamente intercambian información hasta que convergen en una respuesta.

ALGORITMOS BROADCAST

En la sección previa, mostramos cómo hacer broadcast de información en una red. En particular, podemos usar un algoritmo probe para diseminar información y un algoritmo probe/echo para reunir o ubicar información.

En la mayoría de las LAN, los procesadores comparten un canal de comunicación común tal como un Ethernet o token ring. En este caso, cada procesador está directamente conectado a cada uno de los otros. De hecho, tales redes con frecuencia soportan una primitiva especial llamada **broadcast**, la cual transmite un mensaje de un procesador a todos los otros. Esto provee una técnica de programación útil.

Sea *P*[1:n] un arreglo de procesos, y *ch*[1:n] un arreglo de canales, uno por proceso. Entonces un proceso *P*[i] hace broadcast de un mensaje *m* ejecutando

broadcast *ch*(*m*)

La ejecución de **broadcast** pone una copia de *m* en cada canal *ch*[i], incluyendo el de *P*[i]. El efecto es el mismo que ejecutar *n* sentencias **send** en paralelo, donde cada una envía *m* a un canal diferente. El proceso *i* recibe un mensaje desde su canal ejecutando el **receive** usual. No asumimos que la primitiva **broadcast** es indivisible. En particular, los mensajes broadcast por dos procesos *A* y *B* podrían ser recibidos por otros procesos en distinto orden.

Podemos usar broadcasts para diseminar o reunir información; por ejemplo, con frecuencia se usa para intercambiar información de estado del procesador en LAN. También podemos usarlo para resolver muchos problemas de sincronización distribuidos. Esta sección ilustra el poder de broadcasts desarrollando una implementación distribuida de semáforos. La base para los semáforos distribuidos (y otros protocolos de sincronización descentralizados) es un ordenamiento total de los eventos de comunicación. Comenzamos mostrando cómo implementar relojes lógicos y luego mostramos cómo usar esos relojes para ordenar eventos.

Relojes lógicos y Ordenamiento de Eventos

Los procesos en un programa distribuido ejecutan acciones locales y acciones de comunicación. Las acciones locales incluyen cosas tales como leer y escribir variables locales. No tienen efecto directo sobre otros procesos. Las acciones de comunicación son enviar y recibir mensajes. Estos afectan la ejecución de otros procesos pues comunican información y son el mecanismo de sincronización básico. Las acciones de comunicación son así los *eventos* significantes en un programa distribuido. Por lo tanto, usamos el término *evento* para referirnos a la ejecución de **send** o **receive**.

Si dos procesos *A* y *B* están ejecutando acciones locales, no tenemos manera de saber el orden relativo en el cual las acciones son ejecutadas. Sin embargo, si *A* envía un mensaje a *B*, entonces la acción **send** en *A* debe suceder antes de la acción **receive** correspondiente en *B*. Si *B* subsecuentemente envía un mensaje al proceso *C*, entonces la acción **send** en *B* debe ocurrir antes del **receive** en *C*. Más aún, dado que el **receive** en *B* ocurre antes de la acción **send** en *B*, hay un ordenamiento total entre las cuatro acciones de comunicación. Así, “ocurre antes” es una relación transitiva entre eventos relacionados.

Hay un ordenamiento total entre eventos que se afectan uno a otro. Pero, hay solo un orden parcial entre la colección entera de eventos en un programa distribuido. Esto es porque las secuencias de eventos no relacionados (por ej, las comunicaciones entre distintos conjuntos de procesos) podría ocurrir antes, después, o concurrentemente con otros.

Si hubiera un único reloj central, podríamos ordenar totalmente las acciones de comunicación dando a cada uno un único timestamp. En particular, cuando un proceso envía un mensaje, podría leer el reloj y agregar su valor al mensaje. Cuando un proceso recibe un mensaje, podría leer el reloj y registrar el tiempo en el cual el evento **receive** ocurrió. Asumiendo que la granularidad del reloj es tal que "ticks" entre cualquier **send** y el correspondiente **receive**, un evento que ocurre antes que otro tendrá un timestamp más temprano. Más aún, si los procesos tienen identidades únicas, entonces podríamos inducir un ordenamiento total, por ejemplo, usando la identidad del proceso más bajo para romper empates si eventos no relacionados en dos procesos tienen el mismo timestamp.

Desafortunadamente, es bastante restrictivo asumir la existencia de un único reloj central. En una LAN, por ejemplo, cada procesador tiene su propio reloj. Si estuvieran perfectamente sincronizados, entonces podríamos usar los relojes locales para los timestamps. Sin embargo, los relojes físicos nunca están perfectamente sincronizados. Existen algoritmos de sincronización de reloj para mantener dos relojes cercanos uno a otro, pero la sincronización perfecta es imposible. Así necesitamos una manera de simular relojes físicos.

Un *reloj lógico* es un contador entero que es incrementado cuando ocurre un evento. Asumimos que cada proceso tiene un reloj lógico y que cada mensaje contiene un timestamp. Los relojes lógicos son entonces incrementados de acuerdo a las siguientes reglas:

(7.10) **Reglas de Actualización de Relojes Lógicos**. Sea *lc* un reloj lógico en el proceso *A*.
(1) Cuando *A* envía o broadcast un mensaje, setea el timestamp del mensaje al valor corriente de *lc* y luego incrementa *lc* en 1.
(2) Cuando *A* recibe un mensaje con timestamp *ts*, setea *lc* al máximo de *lc* y *ts*+1 y luego incrementa *lc* en 1.

Dado que *A* incrementa *lc* después de cada evento, cada mensaje enviado por *A* tendrá un timestamp diferente y creciente. Dado que un evento **receive** setea *lc* para que sea mayor que el timestamp del mensaje recibido, el timestamp en cualquier mensaje enviado subsecuentemente por *A* tendrán un timestamp mayor.

Usando relojes lógicos, podemos asociar un valor de reloj con cada evento como sigue. Para un evento **send**, el valor del reloj es el timestamp del mensaje, es decir, el valor local de *lc* al comienzo del envío. Para un evento **receive**, el valor del reloj es el valor de *lc* después de ser seteado al máximo de *lc* y *ts*+1 pero antes de que sea incrementada por el proceso receptor. Las reglas anteriores aseguran que si el evento *a* ocurre antes que el evento *b*, entonces el valor del reloj asociado con *a* será más chico que el asociado con *b*. Esto induce un ordenamiento parcial en el conjunto de eventos relacionados en un programa. Si cada proceso tiene una identidad única, entonces podemos tener un ordenamiento total entre todos los eventos usando la identidad menor de proceso como tie-breaker en caso de que dos eventos tengan el mismo timestamp.

Semáforos Distribuidos

Los semáforos son implementados normalmente usando variables compartidas. Sin embargo, podríamos implementarlos en un programa basado en mensajes usando un proceso server (monitor activo). También podemos implementarlos en una manera distribuida sin usar un coordinador central. Aquí se muestra cómo.

Recordemos la definición básica de semáforos: en todo momento, el número de operaciones **P** completadas es a lo sumo el número de operaciones **V** completadas más el valor inicial. Así, para implementar semáforos, necesitamos una manera de contar las operaciones **P** y **V** y una manera de demorar las operaciones **P**. Además, los procesos que "comparten" un semáforo necesitan cooperar para mantener el invariante aunque el estado del programa esté distribuido.

Podemos cumplir estos requerimientos haciendo que los procesos broadcast mensajes cuando quieren ejecutar operaciones **P** y **V** y que examinen los mensajes que reciben para determinar cuando continuar. En particular, cada proceso tiene una cola de mensajes local *mq* y un reloj lógico *lc*. Para simular la ejecución de **P** o **V**, un proceso broadcast un mensaje a todos los procesos usuario, incluyendo él mismo. El mensaje contiene la identidad del emisor, un tag (P o V) y un timestamp. El timestamp en cada copia del mensaje es el valor corriente de *lc*, el cual es modificado de acuerdo a (7.10).

Cuando un proceso recibe un mensaje P o V, almacena el mensaje en su cola de mensajes *mq*. Esta cola es mantenida ordenada en orden creciente de los timestamps de los mensajes. Asumimos por el momento que cada proceso recibe todos los mensajes que fueron broadcast en el mismo orden y en orden creciente de timestamps. Entonces cada proceso sabría exactamente el orden en el cual fueron enviados los mensajes P y V. Así, cada uno podría contar el número de operaciones **P** y **V** y mantener el invariante del semáforo.

Desafortunadamente, es irrealista asumir que **broadcast** es una operación atómica. Dos mensajes broadcast por dos procesos distintos podrían ser recibidos por otros en distinto orden. Más aún, un mensaje con un timestamp menor podría ser recibido después que un mensaje con un timestamp mayor. Sin embargo, distintos mensajes broadcast por un proceso serán recibidos por los otros procesos en el orden en que fueron enviados; estos mensajes tendrán también timestamps crecientes. Esto es porque (1) la ejecución de **broadcast** es la misma que la ejecución concurrente de **send** (que asumimos que provee entrega ordenada y confiable) y (2) un proceso incrementa su reloj lógico después de cada evento de comunicación.

El hecho de que mensajes consecutivos enviados por cada proceso tienen timestamps distintos nos da una manera de tomar decisiones de sincronización. Supongamos que una cola *mq* de mensajes de un proceso contiene un mensaje *m* con timestamp *ts*. Entonces, una vez que el proceso ha recibido un mensaje con un timestamp más grande de cada uno de los otros procesos, se asegura que nunca verá un mensaje con un timestamp menor. En este punto, el mensaje *m* se dice que está *totalmente reconocido* (*fully acknowledged*). Más aún, una vez que *m* está totalmente reconocido, entonces cualquier otro mensaje anterior a éste en *mq* también estará totalmente reconocido pues todos tendrán timestamps menores. Así, la parte de *mq* que contiene mensajes totalmente reconocidos es un *prefijo estable*: ningún mensaje nuevo será insertado en él.

Cada vez que un proceso recibe un mensaje P o V, enviará un mensaje de acknowledgement (ACK). Los ACKs son broadcast para que todos los procesos los vean. Los mensajes ACK tienen timestamps como es usual, pero no son almacenados en las colas de mensajes y no son acknowledged a sí mismos. Simplemente son usados para determinar cuándo un mensaje regular en *mq* se convirtió en totalmente reconocido.

Para completar la implementación de semáforos distribuidos, cada proceso simula las operaciones semáforo. Usa una variable local *sem* para representar el valor del semáforo. Cuando un proceso toma un mensaje ACK, actualiza el prefijo estable de su cola *mq*. Para cada mensaje V, el proceso incrementa *sem* y borra el mensaje V. Luego examina los mensajes P en orden de timestamp. Si *sem* > 0, el proceso decrementa *sem* y borra el mensaje P. En síntesis, cada proceso mantiene el siguiente predicado, que es su loop invariant:

DSEM: sem ≥ 0 $\wedge$ mq totalmente ordenada por timestamps de los mensajes

Los mensajes P son procesados en el orden en el cual aparecen en el prefijo estable de modo que cada proceso hace la misma decisión acerca del orden en el cual se completan las operaciones **P**. Aunque los procesos podrían estar en distintas etapas al manejar los mensajes P y V, cada uno manejará mensajes totalmente reconocidos en el mismo orden.

El algoritmo para semáforos distribuidos es el siguiente:

```
type kind = enum(V,P,ACK)
chan semop[1:n](sender,kind,timestamp : int), go[1:n](timestamp : int)
User[i:1..n]:: var lc : int := 0     # reloj lógico
               var ts : int          # timestamp en mensajes go
               # ejecuta una operación V
               broadcast semop(i,V,lc); lc := lc + 1
```

```
        .....
        # ejecuta una operación P
        broadcast semop(i,P,lc); lc := lc + 1
        receive go[i](ts); lc := max(lc, ts+!); lc := lc + 1
        .....
Helper[i:1..n]:: var mq : queue of (int, kind, int)     # orden por timestamp
        var lc : int := 0
        var sem : int := valor inicial
        var sender : int, k : kind, ts : int
        do true →       {DSEM}
            receive semop[i](sender,k,ts)
            lc := max(lc, ts+1); lc := lc + 1
            if k = P or k = V →
                    insert(sender,k,ts) en el lugar apropiado en mq
                    broadcast semop(i,ACK,lc); lc := lc + 1
            □ k = ACK →
                    registrar que otro ACK fue visto
                    fa fully acknowledged V messages →
                        remover el mensaje de mq; sem := sem + 1
                    af
                    fa fully acknowledged P messages st sem > 0 →
                        remover el mensaje de mq; sem := sem + 1
                        if sender = i → send go[i](lc); lc := lc + 1 fi
                    af
            fi
        od
```

Los procesos *User* inician las operaciones **V** y **P** broadcasting mensajes por los canales *semop*. Los procesos *Helper* implementan las operaciones **V** y **P**. Hay un *Helper* para cada *User*. Cada *Helper* recibe mensajes de su canal *semop*, maneja su cola de mensajes local, broadcast mensajes ACK, y le dice a su proceso *User* cuándo puede seguir luego de una operación **P**. Cada proceso mantiene un reloj lógico, el cual usa para poner timestamps en los mensajes.

Podemos usar semáforos distribuidos para sincronizar procesos en un programa distribuido esencialmente de la misma manera que usamos semáforos regulares en programas de variables compartidas.

Cuando se usan algoritmos broadcast para tomar decisiones de sincronización, cada proceso debe participar en cada decisión. En particular, un proceso debe oir de cada uno de los otros para determinar cuándo un mensaje está totalmente reconocido. Esto significa que estos algoritmos no son buenos para interacciones entre un gran número de procesos. También que deben ser modificados para contemplar fallas.

ALGORITMOS TOKEN-PASSING

Esta sección ilustra otro patrón de comunicación: token-passing entre procesos. Un token es una clase especial de mensaje que puede ser usado o para dar permiso para tomar una acción o para reunir información de estado global. Ilustramos token-passing presentando soluciones a dos problemas adicionales de sincronización. Primero presentamos una solución distribuida simple al problema de la SC. Luego desarrollamos dos algoritmos para detectar cuándo una computación distribuida ha terminado. Token-passing también es la base para otros algoritmos; por ej, para sincronizar el acceso a archivos replicados.

Exclusión Mutua Distribuida

Aunque el problema de la SC reside primariamente en programas de variables compartidas, también se da en programas distribuidos cuando hay un recurso compartido que puede usar a lo sumo un proceso a la vez. Más aún, el problema de la SC con frecuencia es una componente de un problema mayor, tal como asegurar consistencia en un archivo distribuido o un sistema de BD.

Una manera de resolver el problema de la SC es emplear un monitor activo que otorga permiso para acceder a la SC. Para muchos problemas, tales como implementar locks en archivos no replicados, esta es la aproximación más simple y eficiente. En el otro extremo, el problema de la SC puede ser resuelto usando semáforos distribuidos, implementado como muestra la sección anterior. Esa aproximación da una solución descentralizada en la cual ningún proceso tiene un rol especial, pero requiere intercambiar un gran número de mensajes para cada operación semáforo pues cada **broadcast** tiene que ser reconocido (ACK).

Aquí resolvemos el problema de una tercera manera usando un token ring. La solución es descentralizada y fair, como una que usa semáforos distribuidos, pero requiere intercambiar pocos mensajes. Además, la aproximación básica puede ser generalizada para resolver otros problemas de sincronización que no son fácilmente resueltos de otras maneras.

Sea *P*[1:n] una colección de procesos “regulares” que contienen SC y SNC. Necesitamos desarrollar entry y exit protocols que esos procesos ejecuten antes y después de su SC. También los protocolos deberían asegurar exclusión mutua, evitar deadlock y demora innecesaria, y asegurar entrada eventual (fairness).

Dado que los procesos regulares tienen otro trabajo que hacer, no queremos que también tengan que circular el token. Así emplearemos una colección de procesos adicionales, *Helper*[1:n], uno por proceso regular. Estos procesos helper forman un anillo. Un token circula entre los helpers, siendo pasado de *Helper*[1] a *Helper*[2], y así hasta *Helper*[n], el cual se lo pasa a *Helper*[1]. Cuando *Helper*[i] recibe el token, chequea para ver si su cliente *P*[i] quiere entrar a su SC. Si no, *Helper*[i] pasa el token. En otro caso, *Helper*[i] le dice a *P*[i] que puede entrar a su SC, luego espera hasta que *P*[i] salga; luego, pasa el token. Así, los procesos cooperan para asegurar que el siguiente predicado es siempre true:

$$\text{DMUTEX: } (\forall\, i: 1 \le i \le n: \text{P[i] en su SC} \Rightarrow \text{Helper[i] tiene el token}) \wedge \text{hay exactamente un token}$$

La solución completa se da a continuación. El token ring es representado por un arreglo de canales *token*, uno por *Helper*. Para este problema, el token en sí mismo no lleva datos, entonces se representa con un mensaje nulo. Los otros canales son usados para comunicación entre los clientes *P*[i] y sus helpers. Cada *Helper*[i] usa **empty** para determinar si *P*[i] quiere entrar a su SC; si es así, *Helper*[i] envía a *P*[i] un mensaje *go*, y luego espera recibir un mensaje *exit*:

```
chan token[1:n]( ), enter[1:n]( ), go[1:n]( ), exit[1:n]( )
Helper[i:1..n]:: do true →      {DMUTEX}
                    receive token[i]( )          # adquiere el token
                    if not(empty(enter[i])) →  # P[i] quiere entrar
                        receive enter[i]( )
                        send go[i]( )
                        receive exit[i]( )
                    fi
                    send token[(i mod n) + 1]( )    # pasa el token
                od
P[i:1..n]:: do true →
                send enter[i]( )     # entry protocol
                receive go[i]( )
                SC
                send exit[i]( )      # exit protocol
                SNC
            od
```

Esta solución es fair, pues el token circula continuamente, y cuando un *Helper* lo tiene, *P*[i] tiene permiso para entrar si quiere hacerlo. Como está programado, el token se mueve continuamente entre los helpers. Esto es lo que sucede en una red física token-ring. Sin embargo, en un token-ring software, probablemente es mejor agregar algún delay en cada helper para que el token se mueva más lentamente en el anillo.

Detección de Terminación en un Anillo

Es trivial detectar cuándo terminó un programa secuencial. También es simple detectar cuándo un programa concurrente terminó en un único procesador: todo proceso está bloqueado o terminó, y no hay operaciones de E/S pendientes. Sin embargo, no es en absoluto fácil detectar la terminación de un programa distribuido. Esto es porque el estado global no es visible a ningún procesador. Más aún, puede haber mensajes en tránsito entre procesadores.

El problema de detectar cuándo una computación distribuida ha terminado puede ser resuelto de varias maneras. Por ejemplo, podríamos usar un algoritmo probe/echo o usar relojes lógicos y timestamps. Esta sección desarrolla un algoritmo token-passing, asumiendo que toda la comunicación entre procesos es a través de un anillo. La próxima sección generaliza el algoritmo para un grafo. En ambos casos, se usa token-passing para significar cambios de estado.

Sean *P*[1:n] los procesos en alguna computación distribuida. Por ahora, sea *ch*[1:n] un arreglo de canales de comunicación. Por ahora, asumimos que la computación es tal que la comunicación interproceso en la computación forma un anillo. En particular, el proceso *P*[i] recibe mensajes solo por el canal *ch*[i] y envía mensajes solo al canal *ch*[(i **mod** n) + 1]. También asumimos que los mensajes desde cada proceso son recibidos por su vecino en el anillo en el orden en el cual fueron enviados.

En cualquier momento, cada proceso *P*[i] está activo u ocioso. Inicialmente, cada proceso está activo. Está ocioso si terminó o está demorado en una sentencia **receive**. Después de recibir un mensaje, el proceso ocioso se vuelve activo. Así, una computación distribuida terminó si se dan las dos condiciones siguientes:

DTERM: cada proceso está ocioso $\wedge$ no hay mensajes en tránsito

Un mensaje está en tránsito si fue enviado pero aún no repartido al canal de destino. La segunda condición es necesaria pues cuando el mensaje es repartido, podría despertar a un proceso demorado.

Nuestra tarea es obtener un algoritmo de detección de terminación en una computación distribuida arbitraria, sujeto solo a la suposición de que los procesos se comunican en un anillo. Claramente la terminación es una propiedad del estado global, el cual es la unión de los estados de los procesos individuales más los contenidos de los canales. Así, los procesos tienen que comunicarse uno con otro para determinar si la computación terminó.

Para detectar terminación, sea que hay un token, el cual es un mensaje especial que no es parte de la comunicación. El proceso que mantiene el token lo pasa cuando se convierte en ocioso. (Si un proceso terminó su computación, está ocioso con respecto a la computación distribuida pero continúa participando en el algoritmo de detección de terminación. En particular, el proceso pasa el token e ignora cualquier mensaje regular que recibe).

Los procesos pasan el token usando el mismo anillo de canales de comunicación que usan en la computación en sí misma. Por ejemplo, *P*[1] pasa el token a *P*[2] enviando un mensaje al canal *ch*[2]. Cuando un proceso recibe el token, sabe que el emisor estaba ocioso cuando lo envió. Más aún, cuando un proceso recibe el token, tiene que estar ocioso pues está demorado recibiendo desde su canal y no se convertirá en activo nuevamente hasta que reciba un mensaje regular que es parte de la computación distribuida. Así, luego de recibir el token, un proceso envía el token a su vecino, luego espera recibir otro mensaje desde su canal.

La pregunta ahora es cómo detectar que al computación completa terminó. Cuando el token hizo un circuito completo del anillo de comunicación, sabemos que cada proceso estaba ocioso en algún punto. Pero cómo puede el que tiene el token determinar si todos los otros procesos todavía están ociosos y que no hay mensajes en tránsito?

Supongamos que un proceso, digamos *P*[1] tiene inicialmente el token. Cuando *P*[1] se pone ocioso, inicia el algoritmo de detección de terminación pasando el token a *P*[2]. Luego de que el token retorna a *P*[1], la computación ha terminado si *P*[1] ha estado *continuamente ocioso* pues primero pasó el token a *P*[2]. Esto es porque el token va a través del mismo anillo que los mensajes regulares, y los mensajes se distribuyen en el orden en que fueron enviados. Así, cuando el token vuelve a *P*[1], no puede haber mensajes regulares encolados o en tránsito. En esencia, el token "limpia" los canales, poniendo todos los mensajes regulares adelante de él.

Podemos hacer el algoritmo y su corrección más preciso como sigue. Primero, asociamos un color con cada proceso: blue para ocioso y red para activo. Inicialmente todos los procesos están activos, entonces están coloreados red. Cuando un proceso recibe el token, está ocioso, entonces se colorea a sí mismo blue y pasa el token. Si el proceso luego recibe un mensaje regular, se colorea red. Así, un proceso que está blue se ha vuelto ocioso, pasó el token, y se mantuvo ocioso pues pasó el token.

Segundo, asociamos un valor con el token indicando cuántos canales están vacíos si *P*[1] está aún ocioso. Sea *token* el valor del token. Cuando *P*[1] se vuelve ocioso, se colorea a sí mismo azul, setea *token* a 0, y luego envía *token* a *P*[2]. Cuando *P*[2] recibe el token, está ocioso y *ch*[2] podría estar vacío. Por lo tanto, *P*[2] se colorea blue, incrementa *token* a 1, y envía el token a *P*[3]. Cada proceso *P*[i] a su turno se colorea blue e incrementa *token* antes de pasarlo.

Estas reglas de token passing son entonces:

{ RING: *P*[1] es blue $\Rightarrow$ (*P*[1] ... *P*[*token*+1] son blue $\wedge$
ch[2] ... *ch*[(*token* **mod** *n*)+1] están vacíos) }
acciones de *P*[1] cuando primero se vuelve ocioso:
color[1] := blue; *token* := 0; **send** *ch*[2](*token*)
acciones de *P*[i:1..n] luego de recibir un mensaje regular:
color[i] := red
acciones de *P*[i:2..n] luego de recibir el token:
color[i] := blue; *token* := *token* + 1; **send** *ch*[(i **mod** n)+1](*token*)
acciones de *P*[1] luego de recibir el token:
if *color*[1] = blue $\rightarrow$ anunciar terminación y halt **fi**
color[1] := blue; *token* := 0; **send** *ch*[2](*token*)

Como indicamos, las reglas aseguran que el predicado RING es un invariante global. La invarianza de RING se desprende del hecho de que si *P*[1] es blue, no ha enviado ningún mensaje regular desde que envió el token, y por lo tanto no hay mensajes regulares en ningún canal hasta donde reside el token. Más aún, todos estos procesos se mantuvieron ociosos desde que pasaron el token. Así, si *P*[1] es aún blue cuando el token volvió a él, todos los procesos son blue y todos los canales están vacíos. Por lo tanto *P*[1] puede anunciar que la computación terminó.

SERVERS REPLICADOS

Hay ejemplos que involucran el uso de servers replicados, es decir, múltiples procesos server que hacen lo mismo. La replicación sirve a uno de dos propósitos. Primero, podemos incrementar la accesibilidad de datos o servicios teniendo más de un proceso que provea el mismo servicio. Estos servers descentralizados interactúan para proveer a los clientes la ilusión de que hay un único servicio centralizado. Segundo, a veces podemos usar replicación para acelerar el encuentro de la solución a un problema dividiendo el problema en subproblemas y resolviéndolos concurrentemente. Esto se hace teniendo múltiples procesos worker compartiendo una bolsa de subproblemas. Esta sección ilustra ambas aplicaciones mostrando primero cómo implementar un archivo replicado y luego desarrollando un algoritmo de cuadratura adaptiva para integración numérica.

Archivos replicados

Una manera simple de incrementar la probabilidad de que un archivo de datos crítico esté siempre accesible es mantener una copia de back-up del archivo en otro disco, usualmente uno en una máquina diferente. El usuario puede hacer esto manualmente realizando periódicamente una copia del archivo. El file system podría mantener la copia automáticamente. En cualquier caso, sin embargo, los usuarios que deseen acceder al archivo deberían conocer si la copia primaria está disponible y, si no, acceder a la copia.

Una tercera aproximación es que el file system provea replicación transparente. En particular, supongamos que hay *n* copias de un archivo de datos. Cada copia es manejada por un proceso server separado, el cual maneja pedidos de clientes para leer o escribir. Cada server provee al cliente una interfase idéntica. Así, un cliente envía un pedido de apertura a cualquier server y luego continúa conversando con ese server, enviándole pedidos de read y write y eventualmente enviando un pedido de close. Los servers interactúan para presentarles a los clientes la ilusión de que hay una única copia del archivo.

Para que los clientes lo vean así, los file servers tienen que sincronizar unos con otros. En particular, tienen que resolver el problema de la *consistencia de archivo*: los resultados de los pedidos de read y write tienen que ser los mismos, independiente de qué copia del archivo se accede. La consistencia es una instancia del problema de lectores/escritores: dos clientes pueden leer el archivo al mismo tiempo, pero un cliente requiere acceso exclusivo cuando escribe. Hay varias maneras de implementar consistencia. En esta sección, asumimos que los archivos enteros deben mantenerse consistentes. Las mismas técnicas pueden usarse para asegurar consistencia a nivel de registros, lo cual es más apropiado para aplicaciones de BD.

Una manera de resolver el problema de consistencia es asegurar que a lo sumo un cliente a la vez puede acceder cualquier copia del archivo. Esto puede ser implementado, por ejemplo, por la solución distribuida al problema de la SC. Cuando un cliente solicita a cualquiera de los servers abrir un archivo, ese server primero interactúa con los otros servers para adquirir acceso exclusivo. Luego el server procesa los pedidos de read y write. Para un pedido de read, el server lee la copia local del archivo. Para un pedido de write, el server actualiza todas las copias del archivo. Cuando el cliente cierra el archivo, el server libera el control exclusivo.

Esta aproximación es por supuesto más restrictiva que lo necesario pues solo un cliente a la vez puede abrir cualquier copia del archivo. Supongamos en cambio que cuando un cliente pide al server abrir un archivo, indica si solo leerá o si leerá y escribirá. Para permitir lectura concurrente, los servers pueden emplear una variación del algoritmo token-passing para exclusión mutua. En particular, sea que hay un token con valor inicial igual al número *ns* de file servers. Entonces, cuando un cliente abre el archivo para lectura, su server espera el token, lo decrementa en 1, lo envía al próximo server (realmente un proceso helper), y luego maneja los pedidos de lectura del cliente. Después que el cliente cierra el archivo, el server incrementa el valor del token la próxima vez que viene por el anillo. Por otra parte, cuando un cliente abre el archivo para escritura, su server espera a que el token tenga valor *ns* y luego lo mantiene mientras maneja los read y write. Después que el cliente cierra el archivo, el server actualiza todas las copias, luego pone el token en circulación.

El problema con este esquema es que los pedidos de escritura nunca serán servidos si hay una corriente continua de pedidos de lectura. Una aproximación algo distinta da una solución fair. En lugar de emplear solo un token, podemos usar *ns* tokens. Inicialmente, cada server tiene un token. Cuando un cliente quiere leer un archivo, su server debe adquirir un token; cuando un cliente quiere escribir, su server debe adquirir los *ns* tokens. Así, cuando un server quiere escribir el archivo, envía un mensaje a todos los otros servers pidiendo sus tokens. Una vez que los reunió, el server maneja la escritura, propagando las actualizaciones a los otros servers. Cuando un server quiere leer el archivo, puede hacerlo inmediatamente si tiene un token. Si no, le pide un token a los otros servers.

Sin embargo este esquema múltiple token tiene dos problemas potenciales. Primero, dos servers podrían casi al mismo tiempo tratar de adquirir todos los tokens. Si cada uno es capaz de adquirir algunos pero no todos, ningún write será ejecutado. Segundo, mientras un server está adquiriendo todos los tokens para escribir, otro server podría pedir un token para leer. Podemos solucionar ambos problemas usando relojes lógicos para poner timestamps en cada pedido de token. Entonces un server da un token si recibe un pedido con un timestamp más temprano que el tiempo en el cual quiso usar el token. Por ejemplo, si dos servers quieren escribir casi al mismo tiempo, el que inició el pedido de escritura más temprano será capaz de reunir los tokens.

Un atributo atractivo de usar múltiples tokens es que residirán en servers activos. Por ejemplo, después que un server reunió todos los tokens para un write, puede continuar procesando pedidos de write hasta que algún otro server requiera un token. Así, si el archivo replicado está siendo muy escrito por un cliente, el overhead de token-passing puede evitarse. Similarmente, si el archivo es mayormente leído, y solo raramente actualizado, los tokens estarán generalmente distribuidos entre los servers, y por lo tanto los pedidos de lectura podrán ser manejados inmediatamente.

Una variación del esquema múltiple token es *weighted voting*. En nuestro ejemplo, un server requiere un token para leer pero *ns* para escribir. Esto en esencia asigna un peso de 1 a la lectura y un peso de *ns* a la escritura. En cambio puede usarse un conjunto distinto de pesos. Sea *rw* el peso de lectura y *ww* el de escritura. Entonces si *ns* es 5, *rw* podría ser seteado a 2, y *ww* a 4. Esto significa que un server debe tener al menos 2 tokens para leer, y al menos 4 para escribir. Cualquier asignación de pesos puede usarse si se cumplen los siguientes dos requerimientos:

* ww > ns / 2 (para asegurar que las escrituras son exclusivas), y
* (rw+ww) > ns (para asegurar que lecturas y escrituras se excluyen mutuamente)

Con weighted voting, no todas las copias del archivo necesitan ser actualizadas cuando se procesa una escritura. Solo es necesario actualizar *ww* copias. Sin embargo, es necesario leer *rw* copias. En particular, cada acción de escritura debe setear un timestamp en las copias que escribe, y una acción de lectura debe usar el archivo con el timestamp más reciente. Leyendo *rw* copias, un lector se asegura que ve al menos uno de los archivos cambiados por la acción de escritura más reciente.

Pasaje de Mensajes Sincrónico

Dado que la sentencia **send** es no bloqueante con AMP, los canales pueden contener un número ilimitado de mensajes. Esto tiene tres consecuencias. Primero, un proceso que envía puede alejarse arbitrariamente de un proceso receptor.. Si el proceso *A* envía un mensaje al proceso *B* y luego necesita estar seguro que *B* lo tomó, *A* necesita esperar recibir un reply desde *B*. Segundo, el reparto de mensajes no es garantizado si pueden ocurrir fallas. Si *A* envía un mensaje a *B* y no recibe un reply, *A* no tiene manera de saber si el mensajes no pudo ser entregado, si *B* falló, o si el reply no pudo ser entregado. Tercero, los mensajes tienen que ser buffereados, y el espacio en buffer es finito en la práctica. Si se envían demasiados mensajes, o el programa fallará o **send** se bloqueará; cualquier consecuencia viola la semántica del AMP.

El pasaje de mensajes sincrónico (SMP) evita estas consecuencias. en particular, tanto **send** como **receive** son primitivas bloqueantes. Si un proceso trata de enviar a un canal, se demora hasta que otro proceso esté esperando recibir por ese canal. De esta manera, un emisor y un receptor sincronizan en todo punto de comunicación. Si el emisor sigue, entonces el mensaje fue entregado, y los mensajes no tiene que ser buffereados. En esencia, el efecto de la comunicación sincrónica es una sentencia de asignación distribuida, con la expresión siendo evaluada por el proceso emisor y luego siendo asignada a una variable en el proceso receptor.

Hay así un tradeoff entre AMP y SMP. Por un lado, SMP simplifica la resolución de algunos problemas y no requiere alocación de buffer dinámica. Por otro lado, es más difícil, como veremos, programar algoritmos heartbeat y broadcast usando SMP.

Este capítulo describe la sintaxis y semántica de SMP y presenta ejemplos que ilustran su uso. La notación es similar a la introducida por Hoare en 1978 en el paper sobre CSP. Uno de los principales conceptos que introdujo Hoare en el artículo es lo que llamamos *comunicación guardada* (o waiting selectivo). La comunicación guardada combina pasaje de mensajes con sentencias guardadas para producir una notación de programa elegante y expresiva. Ilustraremos la comunicación guardada con SMP. También puede usarse con rendezvous o AMP.

NOTACION

Para evitar confusión con las primitivas **send** y **receive** usadas para AMP, usaremos una notación y terminología distinta para SMP. Dado que la comunicación entre procesos es altamente acoplada, usaremos un esquema directo de nombrado de canal. En particular, cada canal será un link directo entre dos procesos en lugar de un mailbox global. Esto no es estrictamente necesario, pero es más eficiente para implementar.

En esta sección, primero describimos e ilustramos las dos sentencias de comunicación, output (send) e input (receive). Luego extendemos las sentencias guardadas para incluir sentencias de comunicación.

Sentencias de Comunicación

Un proceso se comunica con dispositivos periféricos por medio de sentencias de entrada y salida. Los procesos que comparten variables se comunican uno con otro por medio de sentencias de asignación. El SMP combina estos dos conceptos: las sentencias de entrada y salida proveen el único medio por el cual los procesos se comunican, y el efecto de la comunicación es similar al efecto de una sentencia de asignación.

Supongamos que el proceso *A* quiere comunicar el valor de la expresión *e* al proceso *B*. Esto se realiza con los siguientes fragmentos de programa:

(8.1) *A*:: ... *B* ! *e* ...
B:: ... *A* ? *x* ...

B ! *e* es una *sentencia de salida* (output statement). Nombra un proceso destino *B* y especifica una expresión *e* cuyo valor es enviado a ese proceso. *A* ? *x* es una *sentencia de entrada* (input statement). Nombra un proceso fuente *A* y especifica la variable *x* en la cual se almacenará el mensaje llegado desde la fuente.

Asumiendo que los tipos de *e* y *x* son los mismos, las dos sentencias anteriores se dice que *matchean*. Una sentencia de entrada o salida demora al proceso ejecutante hasta que otro proceso alcance una sentencia de matching. Luego las dos sentencias son ejecutadas simultáneamente. El efecto es asignar el valor de la expresión de la sentencia de salida a la variable de la sentencia de entrada. La ejecución de sentencias de comunicación matching puede verse como una *asignación distribuida* que transfiere un valor de un proceso a una variable de otro. Los dos procesos son sincronizados mientras la comunicación tiene lugar, luego cada uno sigue independientemente.

El ejemplo anterior emplea las formas más simples de sentencias de entrada y salida. Las formas generales son:

Destino ! *port*(e_1, ..., e_n)
Fuente ? *port*(x_1, ..., x_n)

Destino y *Fuente* nombran un único proceso, como en (8.1), o un elemento de un arreglo de procesos. El *port* nombra un canal de comunicación simple en el proceso destino o un elemento de un arreglo de ports en el proceso destino. Las expresiones e_i en una sentencia de salida son enviadas al port nombrado del proceso destino. Una sentencia de entrada recibe un mensaje sobre el port designado desde el proceso fuente y asigna los valores a las variables locales x_i. Los ports son usados para distinguir entre distintas clases de mensajes que un proceso podría recibir. Sin embargo, no declararemos explícitamente los nombres de ports; simplemente los introduciremos cuando sea necesario y les daremos nombres únicos para evitar confusión con otros identificadores.

Dos procesos se comunican cuando ejecutan sentencias de comunicación matching. Informalmente, una sentencia de salida y una de entrada matchean si todas las partes son compatibles. La definición formal es la siguiente:

(8.2) **Sentencias de comunicación matching**. Una sentencia de entrada y una de salida matchean si se dan las siguientes condiciones:

a) La sentencia de salida aparece en el proceso nombrado por la sentencia de entrada
b) La sentencia de entrada aparece en el proceso nombrado por la sentencia de salida
c) Los identificadores de port son los mismos, y, si los hay, los valores subscriptos son los mismos
d) Todas las $x_i := e_i$ son sentencias de asignación válidas, donde las x_i son las variables de la sentencia de entrada y las e_i las expresiones en la sentencia de salida

Emplearemos dos formas abreviadas de sentencias de comunicación. Primero, omitiremos el port cuando no sea importante distinguir entre distintas clases de mensajes. Segundo, si hay una sola expresión o variable y no hay port, omitiremos los paréntesis (empleamos estas abreviaciones en (8.1)).

Como ejemplo, el siguiente proceso filtro copia repetidamente caracteres recibidos de un proceso, *West*, a otro proceso, *East*:

```
Copy:: var c : char
       do true → West ? c; East ! c od
```

En cada iteración, *Copy* entra un carácter desde *West*, luego saca ese carácter por *East*. Cualquiera de estas sentencias se demorará si el otro proceso no está listo para ejecutar una sentencia de matching.

Como segundo ejemplo, el siguiente proceso server computa el máximo común divisor de dos enteros positivos *x* e *y*:

```
GCD:: var x, y : int
    do true → Client ? args(x,y)
            do x > y → x := x - y
             □ x < y → y := y - x
            od
            Client ! result(x)
    od
```

GCD espera recibir entrada en su port *args* desde un único proceso cliente (luego extenderemos este ejemplo para soportar múltiples clientes). *GCD* luego computa la respuesta usando el algoritmo de Euclid y envía la respuesta al port *result* del cliente. *Client* se comunica con *GCD* ejecutando:

```
... GCD ! args(v1, v2); GCD ? result(r) ...
```

Aquí los nombres de ports no son realmente necesarios; sin embargo, ayudan para indicar el rol de cada canal.

Comunicación Guardada

Las sentencias de comunicación habilitan a los procesos para intercambiar valores sin emplear variables compartidas. Sin embargo, por sí mismas, son algo limitadas. Con frecuencia un proceso se quiere comunicar con más de uno de otros procesos (quizás por distintos ports) y no sabe el orden en el cual los otros procesos podrían querer comunicarse con él. Por ejemplo, consideremos extender el proceso *Copy* anterior para que sean buffereados hasta 10 caracteres. Si más de 1 pero menos de 10 caracteres están buffereados, *Copy* puede o ingresar otro carácter desde *West* o sacar otro carácter hacia *East*. Sin embargo, *Copy* no puede saber cuál de *West* e *East* será el próximo en alcanzar una sentencia matching. Como segundo ejemplo, en general un proceso server tal como *GCD* tendrá múltiples clientes. Nuevamente, *GCD* no puede saber cuál cliente será el próximo en querer comunicarse con él.

La comunicación no determinística es soportada en forma elegante extendiendo las sentencias guardadas para incluir sentencias de comunicación. Una *sentencia de comunicación guardada* tiene la forma general:

B; *C* → *S*

Aquí *B* es una expresión booleana opcional, *C* es una sentencia de comunicación opcional, y *S* es una lista de sentencias. Si *B* se omite, tiene el valor implícito de true. Si *C* se omite, una sentencia de comunicación guardada es simplemente una sentencia guardada.

Juntos, *B* y *C* forman la guarda. La guarda *tiene éxito* (*succeds*) si *B* es true y ejecutar *C* no causaría una demora; es decir, algún otro proceso está esperando en una sentencia de comunicación matching. La guarda *falla* si *B* es falsa. La guarda se *bloquea* si *B* es true pero *C* no puede ser ejecutada sin causar demora.

Las sentencias de comunicación guardadas aparecen dentro de sentencias **if** y **do**. Ahora una sentencia **if** es ejecutada como sigue. Si al menos una guarda tiene éxito, una de ellas es elegida no determinísticamente. Primero se ejecuta la sentencia de pasaje de mensajes, y luego la correspondiente lista de sentencias. Si todas las guardas fallan, el **if** termina. Si ninguna guarda tiene éxito y algunas guardas están bloqueadas, la ejecución se demora hasta que alguna guarda tenga éxito. Dado que las variables no son compartidas, el valor de una expresión booleana en una guarda no puede cambiar hasta que el proceso ejecute sentencias de asignación. Así, una guarda bloqueada no puede tener éxito hasta que algún otro proceso alcanza una sentencia de comunicación matching.

Una sentencia **do** es ejecutada de manera similar. La diferencia es que el proceso de selección se repite hasta que todas las guardas fallan.

Nótese que tanto sentencias de entrada como de salida pueden aparecer en guardas. También, podría haber varios pares matching en distintos procesos. En este caso, cualquier par de guardas que tiene éxito podría ser elegido. Se da un ejemplo al final de esta sección.

Como ilustración de la comunicación guardada, podemos reprogramar la versión previa de *Copy* como sigue:

```
Copy:: var c : char
       do West ? c → East ! c od
```

Aquí, la sentencia de entrada está en la guarda. El efecto es el mismo que antes: *Copy* se demora hasta que recibe un carácter desde *West*, luego saca el carácter por *East*.

Usando comunicación guardada, también podemos reprogramar *Copy* para implementar un buffer limitado. Por ejemplo, lo siguiente bufferea hasta 10 caracteres:

```
Copy:: var buffer[1:10] : char
       var front := 1, rear := 1, count := 0
       do count < 10; West ? buffer[rear] →
              count := count + 1; rear := (rear mod 10) + 1
        □ count > 0; East ! buffer[front] →
              count := count - 1; front := (front mod 10) + 1
       od
```

Nótese que la interfase a *West* (el productor) e *East* (el consumidor) no cambió. Esta versión de *Copy* emplea dos sentencias de comunicación guardadas. La primera tiene éxito si hay lugar en el buffer y *West* está listo para sacar un carácter; la segunda tiene éxito si el buffer contiene un carácter e *East* está listo para tomarlo. La sentencia **do** no termina nunca pues ambas guardas nunca pueden fallar al mismo tiempo (al menos una de las expresiones booleanas en las guardas es siempre true).

La última versión de *Copy* ilustra el poder expresivo de la comunicación guardada junto con la comunicación sincrónica: No es necesario para *East* pedir un carácter pues *Copy* puede esperar a darle uno a *East* al mismo tiempo que espera recibir otro carácter desde *West*. La solución también ilustra un tradeoff fundamental entre comunicación asincrónica y sincrónica: con comunicación asincrónica, los procesos como *West* e *East* pueden ejecutar a su propia velocidad pues el buffering es implícito; con comunicación sincrónica, es necesario programar un proceso adicional para implementar buffering si es necesario. El buffering explícito es generalmente mucho menos eficiente que el implícito pues el proceso adicional introduce context switching adicional. Sin embargo, con frecuencia el buffering no se necesita. Por ejemplo, un cliente con frecuencia necesita esperar un reply desde un server después de enviar un pedido; en este caso, no interesa si el pedido es buffereado en un canal o no es repartido hasta que el server lo quiera.

Con frecuencia será útil tener varias sentencias de comunicación guardadas que referencian arreglos de procesos o ports y que difieren solo en el subíndice que emplean. En este caso, precederemos una guarda con un rango especificando los valores de los subíndices. Por ejemplo, podemos reprogramar el proceso server *GCD* como sigue para ser usado por cualquier número de clientes:

```
GCD:: var x, y : int
      do (i:1..n) Client[i] ? args(x,y) →
             do x > y → x := x - y  □ x < y → y := y - x  od
             Client[i] ! result(x)
      od
```

El rango del **do** loop externo es una abreviatura para una serie de sentencias de comunicación guardadas, una para cada valor de *i*. *GCD* espera recibir entrada por su port *args* desde cualquiera de los *n* clientes. En el cuerpo del **do** loop externo, *i* es el índice de ese cliente; *GCD* usa *i* para dirigir los resultados al cliente apropiado.

Este programa para *GCD* ilustra las diferencias entre las notaciones que estamos usando para AMP y SMP. Hay una correspondencia uno a uno entre las sentencias **receive** y las sentencias de entrada, y entre las **send** y las sentencias de salida. Sin embargo, con AMP, *args* y *results* serían arreglos de canales globales, y un proceso cliente tendría que pasar su índice a *GCD*. Aquí los canales son ports asociados con procesos, y el índice del cliente es seteado implícitamente en la sentencia guardada. También, la semántica de las dos notaciones son diferentes. Como veremos, estas diferencias afectan la forma de escribir los programas.

Como ejemplo final, lo siguiente ilustra la elección no determinística que ocurre cuando hay más de un par de guardas matching:

A:: **var** x,y : **int**; **if** *B* ! x → *B* ? y □ *B* ? y → *B* ! x **fi**
B:: **var** x,y : **int**; **if** *A* ! y → *A* ? x □ *A* ? x → *A* ! y **fi**

Esto causa que *A* envíe su valor de *x* a *B*, y *B* envíe su valor de *y* a *A*, pero no especifica el orden en el cual se comunican. O *A* envía a *B* y luego recibe de *B*, o viceversa. Por supuesto, podríamos programar este intercambio de valores más simplemente, por ejemplo, haciendo que *A* primero envíe *x* a *B* y luego reciba *y* desde *B*. Sin embargo, el algoritmo resultante sería asimétrico, mientras que este es simétrico.

Ejemplo

Consideremos el problema de la SC. Sea *C*[1:n] un conjunto de procesos que tienen secciones críticas de código. Estos procesos clientes interactúan con un server, *Sem*, como sigue:

```
C[1:n]:: do true → Sem ! P( )      # entry protocol
                   SC
                   Sem ! V( )      # exit protocol
                   SNC
         od
Sem:: do (i:1..n) C[i] ? P( ) →    #  espera un P de cualquier cliente
                C[i] ? V( )        # espera un V de ese cliente
      od
```

Como su nombre lo indica, *Sem* implementa un semáforo binario. Lo hace recibiendo alternativamente una señal *P* desde cualquier cliente y luego una señal *V* desde el cliente que envió la señal *P*. El valor del semáforo no necesita ser almacenado pues hay un solo semáforo y su valor oscila entre 1 y 0. De hecho, todas las sentencias de entrada y salida en los procesos podrían ser invertidas. Dado que el pasaje de mensajes es sincrónico y los procesos intercambian solo señales, no datos, no interesa en qué dirección fluyen las señales de sincronización.

REDES DE FILTROS

Las redes de filtros son programadas con SMP de manera similar a lo hecho con AMP. Vimos cómo implementar un simple filtro *Copy*. De hecho, tal filtro podría ser puesto entre cualquier par de ports de salida y entrada para simular pasaje de mensajes buffereado. Así, la diferencia esencial entre usar SMP versus AMP es la ausencia de buffering implícito. Esto puede ocasionalmente ser usado para beneficio pues el emisor de un mensaje sabe cuándo fue recibido. La ausencia de buffering puede también complicar algunos algoritmos. Por ejemplo, en un algoritmo heartbeat, puede resultar deadlock si dos procesos vecinos tratan ambos primero de enviar uno al otro y luego de recibir uno del otro.

Esta sección desarrolla soluciones paralelas a dos problemas: generación de números primos y multiplicación matriz/vector. Ambas soluciones emplean redes de filtros. El primer algoritmo usa un arreglo de procesos, donde cada proceso se comunica con sus dos vecinos. El segundo algoritmo usa una matriz de procesos, y cada proceso se comunica con sus 4 vecinos. Como siempre sucede con redes de filtros, la salida de cada proceso es una función de su entrada. Además, los datos fluyen a través de la red. En particular, no hay feedback en el cual un proceso envía datos a otro proceso desde el cual podría haber recibido datos. La próxima sección presenta varios programas paralelos en el cual los procesos intercambian datos.

Generación de Números Primos: La Criba de Eratóstenes

La criba de Eratóstenes es un algoritmo clásico para determinar cuáles números en un rango son primos. Supongamos que queremos generar todos los primos entre 2 y *n*. Primero, escribimos una lista con todos los números:

2 3 4 5 6 7 ... *n*

Comenzando con el primer número no tachado en la lista, 2, recorremos la lista y borramos los múltiplos de ese número. Si *n* es impar, obtenemos la lista:

2 3 5 7 ... *n*

En este momento, los números borrados no son primos; los números que quedan todavía son candidatos a ser primos. Pasamos al próximo número, 3, y repetimos el anterior proceso borrando los múltiplos de 3. Si seguimos este proceso hasta que todo número fue considerado, los números que quedan en la lista final serán todos los primos entre 2 y *n*.

El siguiente programa secuencial implementa el algoritmo.

```
var num[2:n] : ([n-1] 0)
var p := 2, i : int
{ I: p es primo ∧ ∀j: 2 ≤ j ≤ (p-1)^2: num[j] = 0 si y solo si j es primo) }
do p * p ≤ n →
    fa i := 2*p to n by p → num[i] := 1 af   # "tacha" los múltiplos de p
    p := p + 1
    do num[p] = 1 → p := p + 1 od      # busca el próximo número no tachado
od
```

La lista es representada por un arreglo, *num*[2:n]. Las entradas en *num* son inicialmente 0, para indicar que los números de 2 a *n* no están tachados; un número *i* es tachado seteando *num*[i] en 1. El loop externo itera sobre los primos *p*, los cuales son elementos de *num* que permanecen en 0. El cuerpo de ese loop primero tacha los múltiplos de *p* y luego avanza *p* al próximo primo. Como indicamos, el invariante para el loop externo dice que si *num*[j] es 0 y *j* está entre 2 y $(p-1)^2$ entonces *j* es primo. El loop termina cuando p^2 es mayor que *n*, es decir, cuando *p* es mayor que *sqrt*(n). El límite superior para *j* en el invariante y la condición de terminación del loop se desprenden del hecho de que si un número *i* no es primo, tiene un factor primo menor que *sqrt*(i).

Consideremos ahora cómo paralelizar este algoritmo. Una posibilidad es asignar un proceso diferente a cada posible valor de *p* y que cada uno, en paralelo, tache los múltiplos de *p*. Sin embargo, esto tiene dos problemas. Primero, dado que asumimos que los procesos se pueden comunicar solo intercambiando mensajes, deberíamos darle a cada proceso una copia privada de *num*, y tendríamos que usar otro proceso para combinar los resultados. Segundo, tendríamos que usar más procesos que primos (aún si *num* pudiera ser compartido). En el algoritmo secuencial, el final de cada iteración del loop externo avanza *p* al siguiente número no tachado. En ese punto, *p* se sabe que es primo. Sin embargo, si los procesos están ejecutando en paralelo, tenemos que asegurar que hay uno asignado a cada posible primo. Rápidamente podemos sacar todos los números pares distintos de 2, pero no es posible sacar números impares sin saber cuáles son primos.

Podemos evitar ambos problemas paralelizando la criba de Eratóstenes de manera distinta. En particular, podemos emplear un pipeline de procesos filtro. Cada filtro en esencia ejecuta el cuerpo del loop externo del algoritmo secuencial. En particular, cada filtro del pipeline recibe un stream de números de su predecesor y envía un stream de números a su sucesor. El primer número que recibe un filtro es el próximo primo más grande; le pasa a su sucesor todos los números que no son múltiplos del primero.

El siguiente es el algoritmo pipeline para la generación de números primos:

```
# Por cada canal, el primer número es primo y todos los otros números
# no son múltiplo de ningún primo menor que el primer número
Sieve[1]:: var p := 2, i : int
        # pasa los número impares a Sieve[2]
        fa i := 3 to n by 2 → Sieve[2] ! i af
Sieve[i:2..L]: var p : int, next : int
            Sieve[i-1] ? p       # p es primo
            do true →
                # recibe el próximo candidato
                Sieve[i-1] ? next
                # pasa next si no es múltiplo de p
                if next mod p ≠ 0 → Sieve[i+1] ! next fi
            od
```

El primer proceso, *Sieve*[1], envía todos los números impares desde 3 a *n* a *Sieve*[2]. Cada uno de los otros procesos recibe un stream de números de su predecesor. El primer número *p* que recibe el proceso *Sieve*[i] es el i-ésimo primo. Cada *Sieve*[i] subsecuentemente pasa todos los otros números que recibe que no son múltiplos de su primo *p*. El número total *L* de procesos *Sieve* debe ser lo suficientemente grande para garantizar que todos los primos hasta *n* son generados. Por ejemplo, hay 25 primos menores que 100; el porcentaje decrece para valores crecientes de *n*.

El programa anterior termina en deadlock. Podemos fácilmente modificarlo para que termine normalmente usando centinelas, como en la red de filtros merge.

Multiplicación Matriz / Vector

Consideremos el problema de multiplicar una matriz *a* por un vector *b*. Por simplicidad, asumiremos que *a* tiene *n* filas y *n* columnas, y por lo tanto que *b* tiene *n* elementos. Nuestra tarea es computar el producto matriz/vector:

```
x[1:n] := a[1:n, 1:n] x b[1:n]
```

Esto requiere computar *n* productos internos, uno por cada fila de *a* con el vector *b*. En particular, el i-ésimo elemento del vector resultado *x* es:

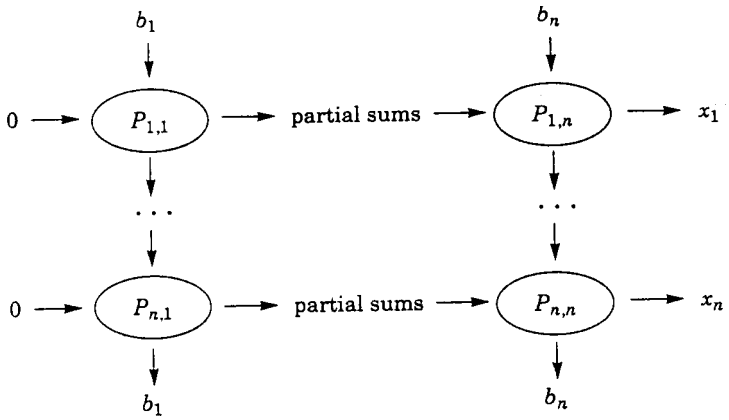
```
x[i] := a[i,1] * b[1] + ... + a[i,n] * b[n]
```

El algoritmo secuencial standard para resolver este problema emplea dos loops. El loop externo itera sobre las filas de *a*; el loop interno computa el producto interno de una fila de *a* y el vector *b*. La complejidad en tiempo de este algoritmo es $O(n^2)$ multiplicaciones. Sin embargo, los *n* productos internos pueden ser computados en paralelo. Esto es porque los elementos de *a* y *b* son solo leídos y los elementos de *x* son disjuntos; por lo tanto los procesos que computan los productos internos no interferirán uno con otro. Si las variables son compartidas, podemos emplear el siguiente algoritmo paralelo:

```
co i := 1 to n →
    x[i] := 0
    fa j := 1 to n → x[i] := x[i] + a[i,j] * b[i] af
oc
```

Nuestra tarea ahora es desarrollar un algoritmo distribuido para resolver este problema usando SMP. Una aproximación sería emplear *n* procesos independientes, donde cada proceso ejecutaría el cuerpo de la sentencia **co** anterior. En este caso, cada proceso debería tener variables locales que contengan una fila de *a* y una copia de *b*.

Una segunda aproximación es emplear una red de n^2 procesos, conectados como muestra la figura:



Cada proceso *P*[i,j] tiene un elemento de *a*, digamos *a*[i,j]. Primero, cada proceso recibe el valor de *b*[i] desde su vecino Norte y lo pasa a su vecino Sur. El proceso luego recibe una suma parcial desde su vecino Oeste, le suma *a*[i,j] * b[i], y envía el resultado a su vecino Este. Los procesos ejecutan lo siguiente:

```
(8.10)   P[i:1..n, j:1..n]:: var sum : real := 0, b : real
                              P[i-1, j] ? b; P[i+1, j] ! b
                              P[i, j-1] ? sum; P[i, j+1] ! (sum + a[i,j] * b)
```

No se muestran los procesos en el borde de la red. Como se muestra en la figura, los proceso en el borde norte envía los elementos de *b*, y los procesos en el borde Oeste envían ceros. Los procesos en el borde Sur solo reciben los elementos de *b*; los procesos en el borde Este reciben los elementos del vector resultado *x*. Estos procesos se necesitan para que las sentencias de comunicación no queden en deadlock.

Los procesos en (8.10) hacen muy poca computación con respecto a la cantidad de comunicación. De este modo, este algoritmo sería muy ineficiente a menos que se implemente directamente en hardware. Sin embargo, podemos usar el mismo patrón de interacción de procesos para realizar multiplicación de dos matrices. Más aún, podemos usar la misma aproximación básica con menos procesos (y procesadores) si particionamos la matriz y el vector en bloques. Por ejemplo, si tenemos 16 procesos y *n* es 1024, entonces podemos particionar *a* en 16 bloques de 256x256 elementos. Cada proceso recibiría 256 elementos de *b* de su vecino Norte y 256 sumas parciales de su vecino Oeste, computaría 256 productos internos parciales, y enviaría los elementos de *b* a su vecino Sur y las sumas parciales a su vecino Este. Con este cambio, cada proceso haría más computación que los de (8.10). Por otro lado, los mensajes serían mayores pues por cada canal deberían transmitirse 256 valores.

PROCESOS PARALELOS INTERACTUANTES

En los algoritmos paralelos de la sección anterior, la información fluía a través de una red de procesos. En esta sección, examinamos algoritmos paralelos en los cuales los procesos intercambian información hasta que han computado una solución.

Primero consideramos una aproximación diferente al sorting paralelo. El algoritmo que presentamos es un ejemplo de algoritmo heartbeat e ilustra cómo programar tales algoritmos usando SMP. El segundo problema es una computación de prefijos paralela para computar todas las sumas parciales de un arreglo. Finalmente se presentan dos algoritmos distintos para multiplicación matriz/matriz. El primero usa un algoritmo broadcast, el segundo usa un algoritmo heartbeat.

Sorting Paralelo: Algoritmo Heartbeat

Consideremos nuevamente el problema de ordenar un arreglo de *n* valores en paralelo. Ya presentamos una solución que empleaba una red de filtros Merge. Aquí se desarrolla una solución diferente, en la cual los procesos vecinos intercambian información repetidamente.

Asumimos que hay dos procesos, *P1* y *P2*, y que cada proceso inicialmente tiene n/2 valores arbitrarios (por simplicidad asumimos que *n* es par). Luego podemos ordenar el conjunto entero de valores en orden no decreciente por el siguiente algoritmo heartbeat. Primero, cada proceso usa un algoritmo secuencial tal como quicksort para ordenar sus *n*/2 valores. Luego los procesos intercambian repetidamente valores hasta que *P*1 tiene los *n*/2 valores menores y *P*2 los *n*/2 mayores. En particular, en cada paso, *P*1 le da a *P*2 una copia de su mayor valor, y *P*2 le da a *P*1 una copia de su valor más chico. Cada proceso inserta el nuevo valor en el lugar apropiado en su lista ordenada de valores, descartando el viejo valor si es necesario. El algoritmo termina cuando el mayor valor de *P*1 no es mayor que el menor valor de *P*2. El algoritmo se garantiza que termina pues cada intercambio (excepto el último) le da dos valores más a los procesos.

Este algoritmo, llamado *compare and exchange*, puede implementarse con el siguiente programa:

```
# Antes de cada intercambio, a1 y a2 están ordenados
# P1 y P2 repetidamente intercambian a1[largest] y a2[smallest]
# hasta que a1[largest] ≤ a2[smallest]
P1:: var a1[1:n/2] : int      # inicializado con n/2 valores
     const largest := n/2; var new : int
     ordenar a1 en orden no decreciente
     P2 ! a1[largest]; P2 ? new    # intercambia valores con P2
     do a1[largest] > new →
          poner new en el lugar correcto en a1, descantando el viejo a1[largest]
          P2 ! a1[largest]; P2 ? new      # intercambia valores con P2
     od
P2:: var a2[1:n/2] : int      # inicializado con n/2 valores
     const smallest := 1; var new : int
     ordenar a2 en orden no decreciente
     P1 ? new; P1 ! a2[smallest]       # intercambia valores con P1
     do a2[smallest] < new →
          poner new en el lugar correcto en a2, descantando el viejo a2[smallest]
          P1 ? new; P1 ! a2[smallest];        # intercambia valores con P1
     od
```

Nótese cómo implementamos los pasos de intercambio. Con AMP, podríamos hacer que cada proceso ejecute un **send** seguido de un **receive**. Sin embargo, dado que las sentencias de entrada y salida son ambas bloqueantes, usamos una solución asimétrica. Para evitar deadlock, *P1* primero ejecuta una salida y luego una entrada, y *P2* ejecuta primero una entrada y luego una salida. Es posible usar comunicación guardada para programar el intercambio simétricamente. En particular, *P1* podría ejecutar:

```
if P2 ? new → P2 ! a1[largest] □ P2 ! a1[largest] → P2 ? new fi
```

Y *P2* podría ejecutar:

```
if P1 ? new → P1 ! a2[smallest] □ P1 ! a2[smallest] → P1 ? new fi
```

Cada proceso espera sacar un valor o ingresar un valor; dependiendo de qué comunicación tiene lugar, el proceso luego completa el intercambio. Esta aproximación sería necesaria si no pudiéramos imponer un orden fijo en el intercambio. Sin embargo, como veremos, la aproximación vista en el programa es mucho más eficiente pues es difícil y costoso implementar comunicación guardada cuando tanto comandos de entrada como de salida aparecen en guardas.

En el mejor caso, los procesos en el programa necesitan intercambiar solo un par de valores. Esto ocurrirá si los n/2 valores menores están inicialmente en P1, y los n/2 mayores en P2. En el peor caso (que ocurrirá si todo valor está inicialmente en el proceso equivocado) los procesos tendrán que intercambiar n/2 + 1 valores: n/2 para tener cada valor en el proceso correcto y uno más para detectar terminación. (En realidad, n/2 intercambios son suficientes, pero no está mal agregar este caso especial al algoritmo).

Ahora consideremos cómo podríamos efectivamente emplear más de dos procesos. En particular, supongamos que usamos *k* procesos *P*[1:k], por ejemplo, porque tenemos *k* procesadores. Inicialmente, cada proceso tiene *n*/*k* valores. Asumimos que ponemos los procesos en una secuencia lineal desde *P*[1] hasta *P*[k] y que cada proceso primero ordena sus *n*/*k* valores. Luego podemos ordenar los *n* elementos usando aplicaciones paralelas repetidas del algoritmo de dos procesos compare-and-exchange.

Cada proceso ejecuta una serie de *rondas*. En las rondas impares, cada proceso con número impar juega el rol de *P1*, y cada proceso con número par el de *P2*. En particular, para todos los valores impares de *i*, *P*[i] intercambia valores con *P*[i+1]. (Si *k* es impar, *P*[k] no hace nada en las rondas impares).

En las rondas pares, cada proceso numerado par juega el rol de *P1*, y cada proceso impar el rol de *P2*. En este caso, para todos los valores pares de *i*, *P*[i] intercambia valores con *P*[i+1]. (En las rondas pares, *P*[1] no hace nada, y, si *k* es par, *P*[k] tampoco hace nada).

Para ver cómo funciona este algoritmo, asumimos que tanto *k* como *n* son 4, y que la lista a ser ordenada es (8,7,6,5). En las rondas impares, los procesos 1 y 2 intercambian valores, así como los procesos 3 y 4. En las rondas pares, solo los procesos 2 y 3 intercambian valores. Para ordenar la lista, los procesos necesitan ejecutar 4 rondas:

ronda	P[1]	P[2]	P[3]	P[4]
0	8	7	6	5
1	7	8	5	6
2	7	5	8	6
3	5	7	6	8
4	5	6	7	8

Nótese que después de la primera ronda los valores mayor y menor fueron movidos hacia su destino final pero los otros dos valores, 6 y 7, se movieron temporariamente en la dirección equivocada. Sin embargo, ningún par de valores es intercambiado en la dirección errónea.

Este algoritmo es llamado *odd/even exchange sort*. Si los procesos comparan e intercambian valores como describimos, después de cada ronda las lista en los procesos que intercambiaron valores estarán ordenadas una en relación a la otra. Más aún, cada intercambio progresa hacia una lista completamente ordenada. Un valor podría moverse temporariamente en la dirección equivocada a donde estará en el estado final, pero solo si es intercambiado con un valor que pertenece al otro lado de él. Así, todos los valores serán ordenados.

La pregunta ahora es: cómo pueden detectar los procesos cuando la lista entera está ordenada? En el algoritmo de dos procesos, cada uno puede rápidamente determinar cuando su mitad de la lista está completa. Lo mismo vale para cada par de procesos en cada ronda del algoritmo k-proceso. Pero un proceso individual no puede detectar que la lista entera está ordenada después de una ronda pues conoce solo dos porciones. De hecho, un proceso podría ejecutar algunas rondas sin ver ningún cambio, y luego podría aparecer un nuevo valor procedente de un proceso a varios links de distancia en la cadena.

Una manera de detectar terminación en el algoritmo k-proceso es emplear un proceso coordinador separado. Después de cada ronda, los *k* procesos le dicen al coordinador si hicieron algún cambio a su porción de la lista. Si ninguno lo hizo, entonces la lista está ordenada. Los procesos luego esperan que el coordinador les diga si terminar o ejecutar otra ronda. Esta aproximación es efectiva pero agrega 2*k mensajes de overhead en cada ronda.

Una segunda aproximación es hacer que cada proceso ejecute suficientes rondas para garantizar que la lista estará ordenada. En el algoritmo 2-proceso, se requiere solo 1 ronda. Sin embargo, en general, es necesario ejecutar al menos *k* rondas, como vimos en el ejemplo anterior de los cuatro procesos. Es complejo mostrar que *k* rondas son suficientes pues los valores pueden moverse temporariamente en contra de su destino final. Pero informalmente, después de k-1 rondas, los n/k valores menores se asegura que están en *P*[1], aún si todos empezaron en *P*[k]. Análogamente, k-1 rondas son suficientes para asegurar que los n/k valores mayores están en *P*[k]. Durante estas k-1 rondas, otros valores quedarán dentro del proceso que es su destino. Así, una ronda más es suficiente para asegurar que todos los valores están en el lugar correcto.

Como describimos el algoritmo k-proceso, cada uno de los *k* procesos intercambia hasta n/k+1 mensajes en cada ronda. Así el algoritmo entero requiere hasta $k^2 * (n/k + 1)$ intercambio de mensajes. Una variación al algoritmo es la siguiente. En cada ronda, cada par de procesos intercambia su conjunto entero de n/k valores. El proceso de número menor luego descarta los n/k valores más grandes, y el proceso de número mayor descarta los n/k valores más chicos. Esto requiere muchos menos mensajes (*k* por ronda), pero cada mensaje es mucho más grande (n/k valores), y cada proceso requiere el doble de almacenamiento local. Otra variación (efectiva en la práctica) es que cada proceso primero intercambie la mitad de sus valores con su vecino. Si no es suficiente, los vecinos luego intercambian otro cuarto de sus valores, luego un octavo, etc.

Computaciones Paralelas de Prefijos

Como describimos en el cap. 3, con frecuencia es útil aplicar una operación a todos los elementos de un arreglo. Por ejemplo, para computar el promedio de un arreglo de valores *a*[1:n], primero necesitamos sumar todos los elementos, luego dividir por *n*. Aquí consideramos nuevamente el problema de computar en paralelo las sumas de todos los prefijos de un arreglo. Esto ilustra cómo programar computaciones paralelas de prefijos usando MP.

Como antes, tenemos un arreglo *a*[1:n] y queremos computar *sum*[1:n], donde *sum*[i] es la suma de los *i* primeros elementos de *a*. Para computar las sumas de todos los prefijos en paralelo, nuevamente podemos usar la técnica de doblar el número de elementos que fueron sumados. Primero, seteamos todos los *sum*[i] a *a*[i]. Luego, en paralelo sumamos *sum*[i-1] a *sum*[i], para todo *i* > 1. En particular, sumamos elementos a distancia 1. Ahora doblamos la distancia, sumando *sum*[i-2] a *sum*[i], en este caso para todo *i* > 2. Si seguimos doblando la distancia, luego de *log*(n) pasos habremos computado todas las sumas parciales.

Para implementar este algoritmo paralelo de prefijos usando MP, necesitamos *n* procesos. Inicialmente, cada uno tiene un valor de *a*. En cada paso, un proceso envía su suma parcial al proceso a distancia *d* a su derecha (si hay alguno), luego espera recibir una suma parcial del proceso a distancia *d* a su izquierda (si lo hay). Los procesos que reciben sumas parciales las suman a *sum*[i]. Luego cada proceso dobla la distancia *d*. El algoritmo completo es el siguiente (el loop invariant *SUM* especifica cuánto del prefijo de *a* sumo cada proceso en cada iteración):

```
Sum[i:1..n]:: var d := 1, sum := a[i], new : int
              { SUM: sum = a[i-d+1] + ... + a[i] }
              do d < n →
                   if (i+d) ≤ n → P[i+d] ! sum fi
                   if (i-d) ≥ 1 → P[i-d] ? new; sum := sum + new fi
                   d := 2 * d
              od
```

Hay dos diferencias esenciales entre esta solución y la dada en el cap. 3. Primero, no hay variables compartidas. Segundo, no hay necesidad de sincronización barrier pues las sentencias de MP fuerzan la sincronización requerida.

Fácilmente podemos modificar este algoritmo para usar cualquier operador binario asociativo. Todo lo que necesitamos cambiar es el operador en la sentencia que modifica *sum*. También podemos adaptar el algoritmo para usar menos de *n* procesos. En este caso, cada proceso tendría un slice del arreglo y necesitaría primero computar las sumas prefijas de ese slice antes de comunicarlas a los otros procesos.

Multiplicación de matrices: Algoritmo Broadcast

La red de filtros usada para multiplicación matriz/vector podría ser extendida para multiplicar una matriz por otra.

Por simplicidad, asumimos que *a* y *b* son matrices cuadradas de *n*x*n*. Nuestra tarea es computar el producto *c* := *a* x *b*. Esto requiere computar n^2 productos internos, uno para cada combinación de fila y columna. Dado que los productos internos son disjuntos, podemos computarlos en paralelo. Así, si *a* y *b* son compartidos, el siguiente algoritmo paralelo computará el producto de matrices:

```
co i := 1 to n, j := 1 to n →
        c[i,j] := 0
        fa k := 1 to n → c[i,j] := c[i,j] + a[i,k] * b[k,j] af
oc
```

Si las matrices *a* y *b* no son compartidas, podríamos multiplicarlas usando n^2 procesos, si cada uno tiene una copia de la fila apropiada de *a* y la columna de *b*. Sin embargo, supongamos inicialmente que *a* y *b* están totalmente distribuidas entre los procesos, con *a*[i,j] y *b*[i,j] almacenados en *P*[i,j]. Como arriba, cada uno de los procesos computa un producto interno. Pero para hacerlo, cada *P*[i,j] primero necesita adquirir los otros elementos de la fila *i* y la columna *j*. Dicho de otra manera, cada *P*[i,j] necesita enviar *a*[i,j] a todos los procesos en la misma fila de *P* y enviar *b*[i,j] a todos los procesos en la misma columna de *P*. Así, necesita broadcast sus valores de *a* y *b* y recibir valores de otros procesos en la misma fila y columna.

Con AMP, ejecutar **broadcast** es equivalente a enviar concurrentemente el mismo mensaje a varios canales. Y **broadcast**, como **send**, es una primitiva no bloqueante. Todas las copias del mensaje se encolan, y cada copia puede ser recibida más tarde por uno de los procesos participantes. Podríamos definir un concepto de **broadcast** para usar con SMP. Pero por la naturaleza bloqueante de los comandos de salida, un **broadcast** sincrónico debería bloquear al emisor hasta que todos los procesos receptores hayan recibido el mensaje. Esto no sería muy útil pues los procesos que necesitan hacer broadcast de mensajes a otros casi siempre necesitan también recibir mensajes de los otros.

Hay dos maneras de obtener el efecto de **broadcast** con SMP. Una es usar un proceso separado para simular un canal broadcast; es decir, los clientes enviarían mensajes broadcast y los recibirían de ese proceso. La segunda manera es usar comunicación guardada. Cuando un proceso quiere tanto enviar como recibir mensajes broadcast, puede usar comunicación guardada con dos guardas. Cada guarda tiene un cuantificador para enumerar los otros partners. Una guarda saca el mensaje hacia todos los otros partners; la otra guarda ingresa un mensaje desde todos los otros partners.

El siguiente programa contiene una implementación distribuida de multiplicación de matrices. Usa el método de comunicación guardada de broadcasting de mensajes. En particular, cada proceso primero envía su elemento de *a* a otros procesos en la misma fila y recibe sus elementos de *a*. Cada proceso luego hace broadcast de su elemento de *b* a todos los procesos en la misma columna y a su turno recibe sus elementos de *b*. (Estas dos sentencias de comunicación guardada podrían ser combinadas en una con 4 guardas). Cada proceso luego computa un producto interno. Cuando el proceso termina, la variable *cij* en el proceso *P*[i,j] contiene el valor resultado *c*[i,j].

```
P[i:1..n, j:1..n]::
   var aij, bij, cij : real
   var row[1:n], col[1:n] : real, k : int
   var sent[1:n] := ([n] false), recvd[1:n] := ([n] false)
   row[j] := aij; col[i] := bij
   # broadcast aij y adquiere otros valores de a[i,*]
   sent[j] := true; recvd[j] := true
   do (k: 1..n) not sent[k]; P[i,k] ! aij → sent[k] := true
     □ (k: 1..n) not recvd[k]; P[i,k] ? row[k] → recvd[k] := true
   od
   # broadcast bij y adquiere otros valores de b[*,j]
```

```
sent := ([n] false); recvd := ([n] false)
sent[i] := true; recvd[i] := true
do (k: 1..n) not sent[k]; P[k,j] ! bij → sent[k] := true
  □ (k: 1..n) not recvd[k]; P[k,j] ? col[k] → recvd[k] := true
od
# computa el producto interno de a[i,*] y b[*,j]
cij := 0
fa k := 1 to n → cij := cij + row[k] * col[k] af
```

En este programa implementamos los intercambios de valores usando sentencias **do** que iteran hasta que cada proceso haya enviado y recibido un valor de los otros n-1 procesos. Para saber cuando terminar cada loop, introdujimos arreglos adicionales booleanos, *sent* y *recvd*, para registrar a cuales procesos se les envió un valor y de cuales se recibió. Esto es necesario pues queremos intercambiar valores exactamente una vez con cada uno de los otros procesos en la misma fila y columna.

En general, **broadcast** tiene que ser implementado como en el programa. Para este problema específico, sin embargo, podríamos implementar el intercambio de mensajes más simplemente haciendo que los procesos circulen sus valores a lo largo de las filas y luego las columnas. En particular, podríamos circular todos los *aij* alrededor de la fila *i* haciendo que cada *P*[i,j] ejecute:

```
next := j; row[j] := aij
fa k := 1 to n-1 →
    if P[i,jΘ1] ! row[next] → P[i,j⊕1] ? row[next⊕1]
    □ P[i,j⊕1] ? row[next⊕1] → P[i,jΘ1] ! row[next]
    fi
    next := next ⊕ 1
af
```

En la primera iteración, P[i,j] da su elemento de *a* su vecino izquierdo y recibe el elemento de *a* de su vecino derecho. En la próxima iteración, P[i,j[pasa ese elemento a su vecino izquierdo y recibe otro elemento de su vecino derecho, el cual fue recibido por este de su vecino derecho en la primera iteración. Los *n* elementos en cada fila de *a* son shifteados circularmente hacia la izquierda *n*-1 veces de modo que cada proceso recibe cada valor de su fila. Los valores pueden ser ciclados a lo largo de las columnas de manera similar.

Multiplicación de Matrices: Algoritmo Heartbeat

Cada proceso en la solución broadcast envía y recibe 2*(n-1) mensajes. Sin embargo, cada proceso almacena una fila entera de *a* y una columna de *b*. Esta sección presenta un algoritmo de multiplicación de matrices que requiere dos mensajes más por proceso, pero cada proceso tiene que almacenar solo un elemento de *a* y uno de *b* a la vez. La solución usa la técnica de shifting introducida al final de la sección anterior. Nuevamente asumimos que inicialmente *a* y *b* están totalmente distribuidos de modo que cada *P*[i,j] tiene los elementos correspondientes de *a* y *b*.

Para computar *c*[i,j], el proceso *P*[i,j] necesita multiplicar cada elemento en la fila *i* de *a* por el correspondiente elemento en la columna *j* de *b*. Pero el orden en el cual *P* realiza estas multiplicaciones no interesa. Esto sugiere que podemos reducir el almacenamiento en cada proceso a un elemento de *a* y uno de *b* a la vez si podemos encontrar una manera de circular los valores entre los procesos para que cada uno tenga un par que necesite en el momento preciso.

Primero consideremos *P*[1,1]. Para computar *c*[1,1], necesita tener cada elemento de la fila 1 de *a* y la columna 1 de *b*. Inicialmente tiene *a*[1,1] y *b*[1,1], de modo que puede multiplicarlos. Si luego shifteamos la fila 1 de *a* hacia la izquierda 1 posición y shifteamos la columna 1 de *b* una posición hacia arriba, *P*[1,1] tendrá *a*[1,2] y *b*[2,1], los cuales puede multiplicar y sumar a *c*[1,1]. Si seguimos shifteando n-1 vez, *P*[1,1] verá todos los valores que necesita.

Desafortunadamente, esta secuencia de multiplicación y shift funcionará solo para los procesos sobre la diagonal. Otros procesos verán todos los valores que necesitan, pero no en las combinaciones correctas. Sin embargo, ves posible reacomodar *a* y *b* antes de empezar la secuencia de multiplicación y shift. En particular, si shifteamos la fila *i* de *a* circularmente a la izquierda *i* posiciones y la fila *j* de *b* circularmente hacia arriba *j* posiciones, entonces cada proceso tendrá los correspondientes pares de valores en cada paso (esta ubicación inicial no es fácil de entender). Podemos ver el resultado del reacomodamiento inicial de los valores de *a* y *b* para una matriz de 4x4:

a[1,2], b[2,1]	a[1,3], b[3,2]	a[1,4], b[4,3]	a[1,1], b[1,4]
a[2,3], b[3,1]	a[2,4], b[4,2]	a[2,1], b[1,3]	a[2,2], b[2,4]
a[3,4], b[4,1]	a[3,1], b[1,2]	a[3,2], b[2,3]	a[3,3], b[3,4]
a[4,1], b[1,1]	a[4,2], b[2,2]	a[4,3], b[3,3]	a[4,4], b[4,4]

El siguiente programa da una implementación del algoritmo de multiplicación de matrices. Cada proceso primero envía su *aij* al proceso *i* columnas a su izquierda y recibe un nuevo *aij* del proceso *i* columnas a la derecha. Luego cada proceso envía su *bij* al proceso *j* filas arriba de él y recibe un nuevo *bij* desde el proceso *j* filas debajo de él. Todas las computaciones de índices de fila y columnas son hechas módulo *n* usando ⊕ y Θ, de modo que los shifts son circulares:

```
P[i:1..n, j:1..n]::
   var aij, bij, cij : real
   var new : real, k : int
   # shiftea valores en aij circularmente i columnas a la izquierda
   if P[i, j Θ i] ! aij → P[i,j ⊕ i] ? aij
    □ P[i, j ⊕ i] ? new → P[i, j Θ i] ! aij; aij := new
   fi
   # shiftea valores en bij circularmente j filas hacia arriba
   if P[i Θ j, j] ! bij → P[i ⊕ ,j, j] ? bij
    □ P[i ⊕ j, j] ? new → P[i Θ j, j] ! bij; bij := new
   fi
   cij := aij * bij
   fa k := 1 to n-1 →
        # shiftea aij uno a la izquierda, bij uno hacia arriba, luego multiplica
        if P[i, j Θ 1] ! aij → P[i,j ⊕ 1] ? aij
         □ P[i, j ⊕ 1] ? new → P[i, j Θ 1] ! aij; aij := new
        fi
        if P[i Θ 1, j] ! bij → P[i ⊕ 1, j] ? bij
        □ P[i ⊕ 1, j] ? new → P[i Θ 1, j] ! bij; bij := new
        fi
        cij := cij + aij * bij
   af
```

Después de reacomodar los aij y bij, los procesos setean cij a aij * bij. Luego ejecutan n-1 fases heartbeat. En cada una, los valores en aij son shifteados 1 lugar a la izquierda, los valores en bij son shifteados un lugar hacia arriba, y luego los nuevos valores son multiplicados y sumados a cij.

Este algoritmo y el broadcast pueden ser adaptados para trabajar sobre particiones de matrices. En particular, podríamos dividir grandes matrices en *k* bloques cuadrados o rectangulares y podríamos hacer que un proceso compute el producto de matrices para cada bloque. Los procesos intercambiarían datos de las mismas maneras que las vistas.

CLIENTES Y SERVIDORES

Dado que la comunicación fluye en una dirección, MP es ideal para programar redes de filtros y peers interactuantes. Como vimos en el cap. 7, MP también puede usarse para programar el flujo de información bidireccional que ocurre en interacciones clientes/servidor.

Esta sección examina varios problemas cliente/servidor e ilustra cómo resolverlos usando SMP. Primero reprogramamos los ejemplos del alocador de recursos y el file server. Ambas soluciones ilustran la utilidad de la comunicación guardada. Luego presentamos dos soluciones MP al problema de los filósofos: una con un manejador de tenedores centralizado y una en la cual el manejo de los tenedores es descentralizado. La solución descentralizada es una instancia interesante de un algoritmo token-passing.

Alocación de Recursos

Recordemos el problema de manejar un recurso múltiple unidad tal como bloques de memoria o archivos. Los clientes piden unidades del recurso a un proceso alocador, las usan, y luego las liberan. Nuevamente asumimos que los clientes adquieren y liberan las unidades una a la vez.

Tal alocador de recursos sirve dos clases de pedidos: adquirir y liberar. Con port naming, podemos usar un port diferente para cada clase de request. En particular, cuando un cliente quiere una unidad, envía un mensaje al port *acquire*. Cuando el pedido puede ser satisfecho, el server aloca una unidad y la retorna al cliente. Luego de usar el recurso, el cliente envía un mensaje al port *release* del server; este mensaje contiene la identidad de la unidad de recurso que el cliente está liberando.

El alocador de recursos y la interfase del cliente se muestran en el siguiente programa. Aquí, usamos comunicación guardada para simplificar mucho el alocador, respecto al del cap. 7. En particular, usamos dos sentencias de comunicación guardadas con cuantificadores. La primera espera mensajes *acquire*, los cuales pueden ser aceptados si hay unidades disponibles. La segunda espera mensajes *release*, los cuales siempre pueden ser aceptados. No hay necesidad de mezclar estas dos clases de mensajes. Más aún, el *Allocator* no necesita salvar pedidos pendientes. Esto es porque puede usar la condición booleana de la primera guarda, *avail* > 0, para diferir la aceptación de mensajes *acquire* cuando no hay unidades libres:

```
Allocator:: var avail := MAXUNITS, units : set of int, unitid : int
            inicializar units a los valores apropiados
            do (c: 1..n) avail  > 0; Client[c]?acquire( ) →
                              avail := avail - 1; unitid := remove(units)
                              Client[c]!reply(unitid)
             □ (c: 1..n) Client[c]?release(unitid) →
                              avail := avail + 1; insert(units,unitid)
            od
Client[i: 1..n]:: var unitid : int
                  Allocator ! acquire( )
                  Allocator ? reply(unitid)
                  # usa el recurso unitid y luego lo libera
                  Allocator ! release(unitid)
                  ....
```

Como está programado, *Allocator* aceptará mensajes *acquire* desde los clientes en algún orden arbitrario. En principio, será el orden en que los clientes enviaron estos mensajes. Sin embargo, si *Allocator* quiere servir pedidos de clientes en algún otro orden (por ej, por menor tiempo de uso del recurso, o por prioridad del cliente) tendríamos que programar *Allocator* como en el cap. 7. Habría que hacer cambios similares para manejar pedidos de múltiples unidades a la vez. En ambos casos, el problema es que las condiciones booleanas en las guardas no pueden depender de los contenidos de un mensaje; solo pueden referenciar variables locales. Así, si un server tiene que mirar un mensaje para determinar si puede ser procesado, debe salvar los pedidos no procesados.

En este programa, los clientes son un arreglo de procesos, y así *Allocator* puede usar un índice de cliente para responderle directamente los mensajes. Esto no es muy satisfactorio en general pues fija tanto el número como los nombres de todos los clientes potenciales. Una aproximación más flexible es usar algún método de nombrado dinámico. Una posibilidad es tener un tipo de datos para identidades de proceso y alguna función por la cual un proceso puede determinar su propia identidad. Otra posibilidad es tener un tipo de datos para los ports. En ambos casos, un cliente pasaría un único valor del tipo apropiado en los mensajes *acquire*; el *Allocator* luego usaría este valor para dirigir un reply al cliente.

File Servers y Continuidad Conversacional

Consideremos nuevamente la interacción entre clientes y file servers descripta en el cap. 7. Como antes, hasta *n* archivos pueden estar abiertos a la vez, y el acceso a cada archivo abierto es provisto por un proceso file server separado. Para usar un archivo, un cliente primero envía un pedido de apertura a cualquiera de los FS y luego espera una respuesta. Subsecuentemente, el cliente entra en una conversación con el server que respondió. Durante la conversación, el cliente envía pedidos de lectura y escritura al server. El cliente finaliza la conversación enviando un pedido de close. Luego el server queda libre para comenzar una conversación con un nuevo cliente.

El siguiente programa muestra cómo los FS y clientes pueden interactuar usando SMP y comunicación guardada. Un pedido de open desde un cliente es enviado a cualquier FS; uno que está libre recibe el mensaje y responde. Tanto el cliente como el FS usan el índice del otro para el resto de la conversación:

```
File[i: 1..n]:: var fname : string, args : otros tipos de argumentos
                var more : bool
                var buffer local, cache, dirección de disco, etc.
                do (c: 1..m) Client[c]?open(fname) →
                   # abre archivo fname; si tiene éxito entonces:
                   Client[c]!open_reply( ); more := true
                   do more →
                        if Client[c]?read(args) →
                            manejar lectura; Client[c]!read_reply(results)
                        □ Client[c]?write(args) →
                            manejar escritura; Client[c]!write_reply(results)
                        □ Client[c]?close( ) →
                            cierra el archivo; more := false
                        fi
                   od
                od
Client[j: 1..m]:: var serverid : int
                   do (i: 1..n) File[i]!open("pepe") →
                        serverid := i; File[i]?open_reply( )
                   od
                  # usa y eventualmente cierra el archivo; por ej, para leer ejecuta:
                   File[serverid]!read(argumentos de acceso)
                   File[serverid]?read_reply(results)
                   .....
```

El patrón de comunicación es idéntico al de la solución con AMP. Los mensajes son enviados a ports en lugar de a canales globales, pero los intercambios de mensaje son los mismos. También, los procesos *File* son casi los mismos. Las diferencias son que la sentencia de entrada para pedidos de *open* fue movida a la guarda del **do** loop externo y que la comunicación guardada es usada en la sentencia **if** que sirve otros pedidos. Estos cambios toman ventajas de la comunicación guardada pero no son necesarios.

El hecho de que los patrones de comunicación son idénticos refleja la similitud entre AMP y SMP para interacción cliente/servidor, pues tal interacción es inherentemente asincrónica. Así, la semántica no bloqueante del **send** no tiene beneficios en este caso.

Filósofos (Centralizado)

En los cap. 4 y 5 discutimos cómo resolver el problema de los filósofos, primero usando semáforos y luego usando CCRs. En esta sección y la siguiente, presentamos dos soluciones que usan MP. Nuevamente representamos los 5 filósofos por procesos. Pero dado que los procesos no pueden compartir variables, tenemos que usar uno o más procesos cuidadores para manejar los cinco tenedores.

Una aproximación es simular la solución de semáforos. En particular, podríamos tener un proceso server que implemente 5 semáforos, uno por tenedor. Este server sería programado como *Sem* en este capítulo. En este caso, un filósofo necesitaría enviar dos mensajes para pedir sus tenedores y dos mensajes para liberarlos. También, tendríamos que asegurar que los filósofos no queden en deadlock.

Una segunda aproximación es simular la solución de CCRs. En este caso, nuevamente usamos un server, pero este mantiene la pista del estado de cada filósofo en lugar del estado de cada tenedor. En particular, el server registra si cada filósofo está comiendo o no. Con esta aproximación, un filósofo necesita enviar solo un mensaje para pedir o liberar ambos tenedores. También es fácil evitar deadlock.

La siguiente solución con MP usa la segunda aproximación. Para pedir permiso para comer, un filósofo envía un mensaje al port *getforks* de *Waiter*. Usando un cuantificador en la sentencia de comunicación guardada que toma señales *getforks*, el *Waiter* puede posponer la aceptación de un pedido desde *Phil*[i] hasta que ningún vecino esté comiendo. Así, la sentencia de salida para *getforks* en *Phil*[i] se bloqueará hasta que el pedido pueda ser atendido. Esto obvia la necesidad de un mensaje de reply explícito desde *Waiter* para decirle al filósofo que puede seguir. Un filósofo libera sus tenedores de manera similar, pero en este caso la sentencia de entrada en *Waiter* no necesita estar guardada. Como se ve en el programa, el loop invariant de *Waiter* es el mismo que el resource invariant de CCRs. El invariante es asegurado por la expresión booleana en la guarda que toma los mensajes *getforks*.

```
Waiter:: var eating[1:5] := ([5] false)
        { EAT: ( ∀ i: 1 ≤ i ≤ 5: eating[i] ⇒ ¬ (eating[iΘ1] ∨ eating[i⊕1]) }
        do (i: 1..5) not (eating [iΘ1] or eating[i⊕1]);
                    Phil[i]?getforks( ) → eating[i] := true
         □ (i: 1..5) Phil[i]?relforks( ) → eating[i] := false
        od
Phil[i: 1..5]:: do true →
                    Waiter ! getforks( )
                    come
                    Waiter ! relforks( )
                    piensa
                od
```

Esta solución no es fair. En particular, un filósofo quedará en inanición si un vecino está comiendo, luego el otro, y así siguiendo. Para tener una solución fair, podríamos hacer algo como tener que el *Waiter* sirva los pedido en orden FCFS. Esto requiere cambiar el cuerpo del *Waiter* y su interfase con los filósofos.

Filósofos (Descentralizado)

El proceso *Waiter* anterior maneja los cinco tenedores. En esta sección desarrollamos una solución descentralizada en la cual hay 5 waiters, uno por filósofo. La solución es otro ejemplo de algoritmo token-passing. En este caso, los tokens son los 5 tenedores. La estructura del patrón de interacción es similar al ejemplo del FS replicado del cap. 7. La solución puede ser adaptada para coordinar el acceso a archivos replicados o para dar una solución eficiente al problema de exclusión mutua distribuida.

Supongamos que hay 5 filósofos, 5 waiters, y 5 tenedores. Los filósofos y waiters son representados por procesos. Los tenedores son tokens; cada tenedor es compartido por dos waiters, uno de los cuales lo tiene a la vez. Cada filósofo interactúa con su propio waiter. Cuando un filósofo quiere comer, le pide a su waiter para adquirir dos tenedores; ese waiter interactúa con los waiters vecinos si es necesario. Luego el waiter mantiene ambos tenedores mientras el filósofo come.

Como con archivos replicados, la clave para una solución distribuida es manejar los tenedores de manera que se evite el deadlock. Idealmente, la solución debería ser fair. Para este problema, podría resultar deadlock si un waiter necesita dos tenedores y no puede tomarlos. Un waiter debe mantener ambos tenedores mientras el filósofo está comiendo. Pero cuando el filósofo no está comiendo, un waiter debería poder ceder sus tenedores. Sin embargo, necesitamos evitar pasar un tenedor de un waiter a otro si no está siendo usado. Por lo tanto necesitamos una manera de que un waiter decida si mantiene un tenedor o lo cede si su vecino lo necesita.

Una manera para que el waiter decida si cede un tenedor o lo mantiene sería usar relojes lógicos y timestamps. En particular, los waiters podrían registrar cuando su filósofo quiere comer, y estos valores de reloj podrían ser usados cuando waiters vecinos están compitiendo por un tenedor. Sin embargo, para este problema hay una solución mucho más simple.

La idea básica para evitar deadlock es que un waiter mantenga un tenedor si es necesario y aún no fue usado; en otro caso el waiter lo cede. Específicamente, cuando un filósofo comienza a comer, su waiter marca ambos tenedores como "sucios". Cuando otro waiter quiere un tenedor, si está sucio y no está siendo usado, el primer waiter lo limpia y lo cede. El primer waiter no puede tomar nuevamente el tenedor hasta que no haya sido usado pues solo son pasados tenedores sucios entre los waiters. Sin embargo, un tenedor sucio puede ser reusado hasta que lo necesite otro waiter.

El algoritmo descentralizado es el siguiente (suele llamarse algoritmo de los "filósofos higiénicos"):

```
Waiter[i: 1..5]::     { EAT: (eating ⇒ haveL ∧ haveR ∧ dirtyL ∧ dirtyR) }
   var eating := false, hungry := false    # estado de Phil[i]
   var haveL, haveR : bool
   var dirtyL := false, dirtyR := false
   if i = 1 → haveL := true; haveR := true
   □ i ≥ 2 and i ≤ 4 → haveL := false; haveR := true
   □ i = 5 → haveL := false; haveR := false
   fi
   do Phil[i]?hungry( ) →
         hungry[i] := true    # Phil[i] quiere comer
   □ hungry and haveL and haveR →
         hungry := false; eating := true   # Phil[i] puede comer
         dirtyL := true; dirtyR := true; Phil[i]!eat( )
   □ hungry and not haveL; Waiter[iΘ1]!need( ) →
         haveL := true          # pidió el tenedor izquierdo; ahora lo tiene
   □ hungry and not haveR; Waiter[i⊕1]!need( ) →
         haveR := true          # pidió el tenedor derecho; ahora lo tiene
   □ haveL and not eating and dirtyL; Waiter[iΘ1]?need( ) →
         haveL := false; dirtyL := false     # cede el tenedor izquierdo
   □ haveR and not eating and dirtyR; Waiter[i⊕1]?need( ) →
         haveR := false; dirtyR := false     # cede el tenedor derecho
   □ Phil[i]?full( ) →
         eating := false          # Phil[i] terminó de comer
   od
Phil[i: 1..5]:: do true →
                   Waiter[i] ! hungry( ); Waiter[i] ? eat( );
                   come
                   Waiter[i] ! full( )
                   piensa
                od
```

Cuando un filósofo quiere comer, envía un mensaje *hungry* a su waiter, luego espera a que éste le mande un mensaje *eat*. Los waiters vecinos intercambian mensajes *need*. Cuando un filósofo está hambriento y su waiter necesita un tenedor, ese waiter saca un mensaje *need* al waiter que tiene el tenedor. Los otros waiters aceptan el mensaje *need* cuando el tenedor está sucio y no está siendo usado (Con AMP no necesitamos mensajes adicionales para pasar tenedores entre waiters). La aserción EAT en *Waiter* indica qué es true cuando un filósofo está comiendo; es el loop invariant del waiter.

Como se ve en el programa, *Waiter*[1] inicialmente tiene dos tenedores, *Waiter*[5] ninguno, y los otros tienen uno cada uno. Todos los tenedores están inicialmente limpios. Es imperativo que o los tenedores estén distribuidos asimétricamente o que algunos estén limpios y otros sucios. Por ejemplo, si cada filósofo inicialmente tiene un tenedor y todos están limpios, podría haber deadlock si todos los waiters quieren los dos tenedores al mismo tiempo. Cada uno cedería uno y luego esperaría por siempre tomarlo nuevamente.

La solución propuesta también es fair, asumiendo que un filósofo no come siempre y que una guarda que tiene éxito es eventualmente elegida. En particular, si un waiter quiere un tenedor que tiene otro, eventualmente lo obtendrá. Si el tenedor está sucio y en uso, eventualmente el otro filósofo terminará de comer, y por lo tanto el otro waiter eventualmente ejecutará la sentencia de comunicación guardada que cede el tenedor. Si el tenedor está limpio, es porque el otro filósofo está hambriento y el otro waiter está esperando para tomar un segundo tenedor. Por razonamiento similar, el otro waiter eventualmente tomará el segundo tenedor, su filósofo comerá, y por lo tanto el waiter cederá el primer tenedor. El otro waiter eventualmente tomará el segundo tenedor porque no hay ningún estado en el cual cada waiter mantiene un tenedor limpio y quiere un segundo (esta es otra razón por la cual es imperativa la inicialización asimétrica).

RPC y Rendezvous

Tanto AMP como SMP son lo suficientemente poderosos para programar las cuatro clases de procesos: filtros, clientes, servidores y peers. Dado que la información fluye a través de canales en una dirección, ambos son ideales para programar filtros y peers. Sin embargo, el flujo de información en dos direcciones entre clientes y servidores tiene que ser programada con dos intercambios de mensajes explícitos usando dos canales diferentes. Más aún, cada cliente necesita un canal de reply diferente; esto lleva a un gran número de canales.

Este capítulo examina dos notaciones de programación adicionales (remote procedure call [RPC] y rendezvous) que son ideales para interacciones cliente/servidor. Ambas combinan aspectos de monitores y SMP. Como con monitores, un módulo o proceso exporta operaciones, y las operaciones son invocadas por una sentencia **call**. Como con las sentencias de salida en SMP, la ejecución de **call** demora al llamador. La novedad de RPC y rendezvous es que una operación es un canal de comunicación bidireccional desde el llamador al proceso que sirve el llamado y nuevamente hacia el llamador. En particular, el llamador se demora hasta que la operación llamada haya sido ejecutada y se devuelven los resultados.

La diferencia entre RPC y rendezvous es la manera en la cual se sirven las invocaciones de operaciones. Una aproximación es declarar un procedure para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada llamado. Esto se llama *remote procedure call* pues el llamador y el cuerpo del procedure pueden estar en máquinas distintas. La segunda aproximación es *rendezvous* con un proceso existente. Un rendezvous es servido por medio de una sentencia de entrada (o accept) que espera una invocación, la procesa, y luego retorna resultados. (Suele llamarse rendezvous extendido para contrastarlo con el rendezvous simple entre sentencias de comunicación matching en SMP).

Este capítulo describe notaciones de programación representativas para RPC y rendezvous e ilustra su uso. Como mencionamos, cada una facilita interacciones cliente/servidor. También pueden usarse para programar filtros y peers interactuantes. Sin embargo, veremos que programar filtros y peers no es sencillo pues ni RPC ni rendezvous soportan comunicación asincrónica. Afortunadamente, podemos resolver este problema combinando RPC, rendezvous, y AMP en un lenguaje poderoso y bastante simple.

Ilustramos el uso de estas tres notaciones y los tradeoffs entre ellas por medio de ejemplos. Algunos examinan problemas ya resueltos para facilitar la comparación de las distintas formas de pasaje de mensajes. Otros ejemplos resuelven nuevos problemas, incluyendo un problema de labeling y un algoritmo paralelo branch-and-bound para resolver el problema del viajante de comercio.

REMOTE PROCEDURE CALL

En el capítulo 6, usamos dos clases de componentes en los programas: procesos y monitores. Los monitores encapsulan variables compartidas; los procesos se comunican y sincronizan llamando a procedures del monitor. También, los procesos y monitores en un programa se asume que están en el mismo espacio de direcciones compartido.

Con RPC, usaremos una componente de programa (el módulo) que contiene tanto procesos como procedures. También, permitiremos que los módulos residan en espacios de direcciones distintos, por ejemplo, en diferentes nodos en una red. Los procesos dentro de un módulo pueden compartir variables y llamar a procedures declarados en ese módulo. Sin embargo, un proceso en un módulo puede comunicarse con procesos en un segundo módulo solo llamando procedures del segundo módulo.

Para distinguir entre procedures que son locales a un módulo y los que proveen canales de comunicación, un módulo tiene dos partes. La parte de especificación (spec) contiene headers de procedures que pueden ser llamados desde otros módulos. El cuerpo implementa estos procedures y opcionalmente contiene variables locales, código de inicialización, y procedures locales y procesos. La forma de un módulo es:

```
module Mname
  headers de procedures visibles
body
  declaraciones de variables
  código de inicialización
  cuerpos de procedures visibles
  procedures y procesos locales
end
```

También usaremos arreglos de módulos, los cuales serán declarados agregando información de rango al nombre de un módulo.

El header de un procedure visible es especificado por una declaración de operación, y tiene la forma:

op *opname* (formales) **returns** result

El cuerpo de un procedure visible es contenido en una declaración **proc**:

```
proc opname (identificadores formales) returns identificador resultado
  variables locales
  sentencias
end
```

Un **proc** es como un procedure, excepto que los tipos de los parámetros y el resultado no necesitan ser especificados.

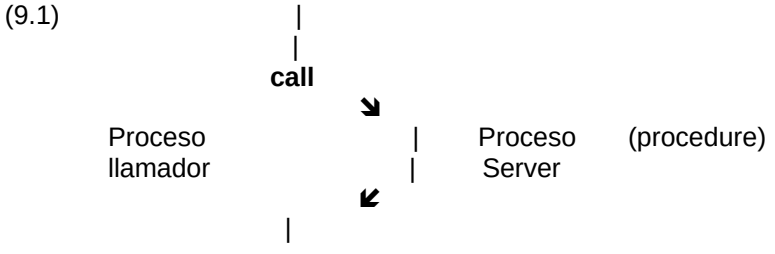
Como con monitores, un proceso (o procedure) en un módulo llama a un procedure en otro ejecutando:

call *Mname.opname* (argumentos)

Para un llamado local, el nombre del módulo puede ser omitido.

La implementación de un llamado intermódulo es distinta que para un llamado local, pues dos módulos pueden estar en espacios de direcciones diferentes. En particular, un nuevo proceso sirve el llamado. El proceso llamador se demora mientras este proceso server ejecuta el cuerpo del procedure que implementa *opname*. Cuando el server vuelve de *opname*, envía los argumentos resultado y cualquier valor de retorno al proceso llamador, luego el server termina. Después de recibir resultados, el proceso llamador continúa. Si el proceso llamador y el procedure están en el mismo espacio de direcciones, con frecuencia es posible evitar crear un nuevo proceso para servir un llamado remoto (el proceso llamador puede temporariamente convertirse en server y ejecutar el cuerpo del procedure. Pero en general, un llamado será remoto, de modo que un proceso server debe ser creado o alocado de un pool preexistente de servers disponibles.

Para ayudar a clarificar la interacción entre el proceso llamador y el proceso server, el siguiente diagrama muestra su ejecución:



Sincronización en Módulos

Por sí mismo, RPC es puramente un mecanismo de comunicación. Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador. Conceptualmente, es como si el proceso llamador mismo estuviera ejecutando el llamado, y así la sincronización entre el llamador y el server es implícita.

También necesitamos alguna manera para que los procesos en un módulo sincronicen con cada uno de los otros. Estos incluyen tanto a los procesos server que están ejecutando llamados remotos como otros procesos declarados en el módulo. Como es habitual, esto comprende dos clases de sincronización: exclusión mutua y sincronización por condición.

Hay dos aproximaciones para proveer sincronización en módulos, dependiendo de si los procesos en el mismo módulo ejecutan con exclusión mutua o ejecutan concurrentemente. Si ejecutan con exclusión (es decir, a lo sumo hay uno activo por vez) entonces las variables compartidas son protegidas automáticamente contra acceso concurrente. Sin embargo, los procesos necesitan alguna manera de programar sincronización por condición. Para esto podríamos usar automatic signalig (**await** B) o variables condición.

Si los procesos en un módulo pueden ejecutar concurrentemente (al menos conceptualmente), necesitamos mecanismos para programar tanto exclusión mutua como sincronización por condición. En este caso, cada módulo es en sí mismo un programa concurrente, de modo que podríamos usar cualquiera de los métodos descriptos anteriormente. Por ejemplo, podríamos usar semáforos dentro de los módulos o podríamos usar monitores locales. De hecho, como veremos, podríamos usar rendezvous. (O usar MP).

Un programa conteniendo módulos es ejecutado primero ejecutando el código de inicialización en cada módulo. El código de inicialización en distintos módulos puede ser ejecutado concurrentemente, asumiendo que este código no hace llamados remotos. Luego los procesos declarados en cada módulo comienzan a ejecutar. Si la exclusión es implícita, entonces un proceso a la vez puede ejecutar en un módulo; cuando se demora o termina, otro puede ejecutar. Si los procesos en módulos ejecutan concurrentemente, todos pueden comenzar la ejecución a la vez.

Si la exclusión es implícita, los módulos son más simples para programar que si los procesos pueden ejecutar concurrentemente. Esto es porque las variables compartidas no necesitan ser protegidas contra acceso concurrente. Si un módulo ejecuta en un único procesador, la implementación puede ser también un poco más eficiente si los procesos ejecutan con exclusión. Esto es porque habrá context switching a la entrada, salida o puntos de demora en los procedures o procesos, no en puntos arbitrarios donde los resultados intermedios de una computación podrían estar en registros.

Por otro lado, es más general asumir que los procesos pueden ejecutar concurrentemente. La ejecución concurrente puede ser mucho más eficiente en un multiprocesador de memoria compartida. Además, con el modelo de ejecución concurrente podemos implementar timeslicing para dividir el tiempo de ejecución entre procesos y para tener mayor control sobre procesos "runaway" (los que están en loops permanentes). Esto no es posible con el modelo de ejecución exclusiva a menos que los procesos cedan el procesador dentro de una cantidad de tiempo razonable. Eso es porque, en ese modelo, podemos switchear el contexto solo cuando el proceso ejecutante alcanza un punto de salida o de demora.

Dado que es más general, asumiremos que los procesos dentro de un módulo ejecutan concurrentemente. Las próximas secciones presentan ejemplos. Por ahora, usamos semáforos para programar exclusión mutua y sincronización por condición; luego agregaremos rendezvous y comunicación guardada.

Time Server

Consideremos el problema de implementar un time server, es decir, un módulo que provee servicios de timing a módulos clientes. Supongamos que el time server define dos operaciones visibles: *get_time* y *delay*(*interval*). Un proceso cliente obtiene el tiempo llamando a *get_time*. Un cliente llama a *delay* para bloquearse durante *interval* unidades de tiempo. El time server también contiene un proceso interno que continuamente inicia un timer hardware, luego incrementa el tiempo cuando ocurre la interrupción de timer.

El siguiente programa muestra el módulo del time server:

```
module TimeServer
  op get_time( ) returns time : int
  op delay( interval : int )
body
  var tod := 0    { tod ≥ 0 y crece monótonamente }
  var m : sem := 1   { (m=1) ⇒ (∀ procesos durmiendo: tod < waketime) }
  var d[1:n] : sem := ([n] 0)       # semáforos de demora privados
  var napQ : queue of (waketime, proces_id : int)
  proc get_time( ) returns time
    time := tod
  end
  proc delay(interval)    # asumimos interval > 0
    var waketime : int := tod + interval
    P(m)
    insert (waketime, myid) en el lugar apropiado de napQ
    V(m)
    P(d[myid])     # espera a ser despertado
  end
  Clock:: do true →
                iniciar hardware timer; esperar interrupción
                P(m)
                tod := tod + 1
                do tod ≥ menor waketime en napQ →
                     remove (waketime, id) de napQ; V(d[id])
                od
                V(m)
          od
end
```

El time of day es almacenado en la variable *tod*. Múltiples clientes pueden llamar a *get_time* y a *delay* al mismo tiempo, y por lo tanto varios procesos podrían estar sirviendo estos llamados concurrentemente. Los procesos que sirven llamados de *get_time* pueden ejecutar concurrentemente pues solo leen el valor de *tod*. Sin embargo, *delay* y *tick* necesitan ejecutar con exclusión mutua cuando están manipulando *napQ*, la cola de procesos cliente "durmiendo". (En *delay*, la asignación a *wake_time* no necesita estar en la SC pues *tod* es la única variable compartida y solo es leída). El valor de *myid* en *delay* se asume único entre 1 y *n*; se usa para indicar el semáforo privado sobre el cual está esperando un cliente. Luego de una interrupción de reloj, *Timer* ejecuta un loop para chequear *napQ*; cuando expiró un intervalo de demora señaliza al semáforo de demora apropiado. Se usa un loop pues más de un proceso pueden estar esperando el mismo tiempo de wakeup.

Caches en un File System distribuido

Consideramos ahora una versión simplificada de un problema que se da en la mayoría de los file systems distribuidos. Supongamos que procesos de aplicación están ejecutando en una workstation y que los archivos de datos están almacenados en un file server. Ignoraremos cómo son abiertos y cerrados los archivos, y solo enfocaremos la lectura y escritura. Cuando un proceso de aplicación quiere acceder un archivo, llama a un procedure *read* o *write* en un módulo local *FileCache*. Las aplicaciones leen y escriben arreglos de caracteres (bytes). A veces una aplicación leerá o escribirá unos pocos caracteres; en otras podría leer o escribir miles.

Los archivos están almacenados en el disco del file server en bloques de 1024 bytes cada uno. El módulo *FileServer* maneja el acceso a bloques de disco. Provee dos operaciones, *readblk* y *writeblk* para leer y escribir bloques enteros.

El módulo *FileCache* mantiene un cache de bloques de datos recientemente leídos. Cuando una aplicación pide leer una parte de un archivo, *FileCache* primero chequea si los bytes pedidos están en su cache. Si es así, rápidamente puede satisfacer el pedido del cliente. Si no, tiene que llamar al procedure *readblk* del módulo *FileServer* para tomar los bloques del disco. (*FileCache* podría también hacer read-ahead si detecta que un archivo está siendo accedido secuencialmente, que es el caso más frecuente).

Los pedidos de escritura se manejan de manera similar. Cuando una aplicación llama a *write*, los datos están almacenados en un bloque en el cache local. Cuando un bloque está lleno o se necesita que satisfaga otro pedido, *FileCache* llama a la operación *writeblk* en *FileServer* para almacenar el bloque en disco. (O *FileCache* podría usar una estrategia write-through, en cuyo caso llamaría a *writeblk* después de cada pedido de write. Usar write-through garantiza que los datos están almacenados en el disco cuando se completa una operación *write*, pero enlentece la operación).

El siguiente programa contiene outlines de estos módulos. Cada uno por supuesto ejecuta en una máquina distinta. Los llamados de aplicación a *FileCache* son de hecho llamados locales, pero los llamados de *FileCache* a *FileServer* son llamados remotos. El módulo *FileCache* es un server para procesos de aplicación; el módulo *FileServer* es un server para varios clientes *FileCache*, uno por workstation:

```
module FileCache            # ubicado en cada WS sin disco
  op read(count : int; res buffer[1:*] : char )
  op write(count : int; buffer[1:*] : char)
body
  var cache de bloques de archivo
  var variables para registrar información de descriptor de archivo
  var semáforos para sincronización de acceso a cache (si es necesario)
  proc read(count, buffer)
    if los datos necesarios no están en cache →
        Seleccionar bloque de cache a usar
        Si es necesario, call FileServer.writeblk( .... )
        call FileServer.readblk( .... )
    fi
    buffer := count bloques apropiados desde el bloque cache
  end
  proc write(count, buffer)
    if bloque apropiado no está en cache →
        Seleccionar bloque de cache a usar
        Si es necesario, call FileServer.writeblk( .... )
    fi
    bloque cache := count bytes desde buffer
  end
end

module FileServer           # ubicado en un file server
  op readblk(fileid, offset : int; res blk[1:1024] : char)
  op writeblk(fileid, offset : int; blk[1:1024] : char)
```

```
body
  var cache de bloques de disco
  var cola de pedidos de acceso a disco pendientes
  var semáforos para sincronizar el acceso a cache y la cola
  proc readblk(fileid, offset, blk)
    if los datos necesarios no están en cache →
          Almacenar el pedido de lectora en la cola del disco
          Esperar que la operación de lectura sea procesada
    fi
    blk := bloque de disco apropiado
  end
  proc writeblk(fileid, offset, blk)
    Seleccionar bloque desde el cache
  Si es necesario almacenar pedido de lectura en la cola de disco y
        esperar que el bloque sea escrito al disco
    bloque cache := blk
  end
  DiskDrive:: do true →
                  Esperar un pedido de acceso al disco
                  Iniciar una operación de disco; esperar una interrupción
                  Despertar al proceso que espera que se complete este pedido
              od
end
```

Asumiendo que hay un módulo *FileCache* por proceso de aplicación, *FileCache* no necesita sincronización interna pues a lo sumo un llamado de *read* o *write* podrían estar ejecutando a la vez. Sin embargo, si los procesos de aplicación usan el mismo módulo *FileCache* (o si el módulo contiene un proceso que implementa read-ahead) entonces *FileCache* necesitaría proteger el acceso al cache compartido.

El módulo *FileServer* requiere sincronización interna pues es compartido por varios módulos *FileCache* y contiene un proceso interno *DiskDriver*. En particular, los servers que manejan llamados a *readblk* y *writeblk* y el proceso *DiskDriver* necesitan sincronizar uno con otro para proteger el acceso al cache de bloques de disco y para hacer scheduling de las operaciones de acceso al disco. (En el programa no se muestra el código de sincronización pues ya se vieron muchos ejemplos de cómo programarlo):

Red de Ordenación de Filtros Merge

Aunque RPC provee buen soporte para programar interacciones cliente/servidor, no es bueno para programar filtros o peers interactuantes. Esta sección reexamina el problema de implementar una red de ordenación usando filtros merge, introducidos en el cap. 7. También introduce una manera de soportar paths de comunicación dinámicos por medio de "capabilities", que son punteros a operaciones en otros módulos.

Recordemos que un merge filter consume dos streams de entrada y produce uno de salida. Cada stream de entrada se asume que está ordenado; la tarea del filtro es mezclar los valores de entrada para producir una salida ordenada. Asumiremos que el final de un stream de entrada está marcado por EOS.

Un problema al usar RPC para programar un merge filter es que RPC no soporta comunicación directa proceso-a-proceso. En lugar de eso, la comunicación entre procesos tiene que ser programada indirectamente usando módulos como buffers de comunicación. En resumen, necesitamos implementar comunicación interproceso explícitamente en nuestro programa; no es provista como una primitiva como en AMP o SMP. El programa resultante ejecutará con la misma eficiencia, pero el programador tiene que escribir más.

Otro problema es cómo unir instancias de filtros merge una con otra. En particular, cada filtro necesita dirigir su stream de salida a uno de los streams de entrada de otro filtro. Sin embargo, los nombres de operación (que proveen los canales de comunicación) son identificadores distintos. Así, cada stream de entrada necesita ser implementado por un procedure distinto . Esto hace difícil usar nombrado estático pues un merge filter necesita saber el nombre literal de la operación a llamar para dar un valor de salida al próximo filtro. Una aproximación mucho mejor es emplear nombrado dinámico, en el cual a cada filtro se le pasa un link a la operación para usar como salida. Representaremos los links dinámicos por *capabilities*, que son esencialmente punteros a operaciones.

El siguiente programa contiene un módulo que implementa un merge filter:

```
optype put(value : int)     # tipo de operaciones para streams de datos
module Merge[1:n]
   op put1 : put, put2 : put, initial(c : cap put)
body
  var in1, in2 : int      # valores de entrada desde los streams 1 y 2
  var out : cap put       # capability para operación de salida
  var empty1 : sem := 1, full1 : sem := 0
  var empty2 : sem := 1, full2 : sem := 0
  proc initial(c)         # llamado primero para tomar el canal de salida
     out := c
  end
  proc put1(v1)           # llamado para producir el próximo valor del stream 1
     P(empty1); in1 := v1; V(full1)
  end
  proc put2(v2)           # llamado para producir el próximo valor del stream 2
     P(empty2); in2 := v2; V(full2)
  end
  M:: P(full1); P(full2)
      do in1 ≠ EOS and in2 ≠ EOS →
         if in1 ≤ in2 → call out(in1); V(empty1); P(full1)
          □ in2 ≤ in1 → call out(in2); V(empty2); P(full2)
         fi
       □ in1 ≠ EOS and in2 = EOS →
         call out(in1); V(empty1); P(full1)
       □ in1 = EOS and in2 ≠ EOS →
         call out(in2); V(empty2); P(full2)
      od
      call out(EOS)
end
```

La primera línea declara el tipo de las operaciones que son usadas para canales de comunicación; en este caso, cada uno tiene un valor entero. Cada módulo provee dos operaciones, *put1* y *put2*, que otros módulos pueden llamar para producir los streams de entrada. Cada uno también tiene una tercera operación, *initial*, que un módulo principal (que no se muestra) llamaría para inicializar la capability *out*. or ejemplo, la rutina principal podría dar a *Merge*[i] una capability para la operación *put2* o *Merge*[j] ejecutando:

```
call Merge[i].initial(Merge[j].put2)
```

Asumimos que cada módulo es inicializado antes de que cualquiera de ellos comience a producir resultados.

El resto del módulo es similar al *Merge* del cap. 7. Las variables *in1* e *in2* corresponden a los canales del mismo nombre del cap;. 7, y el proceso *M* simula las acciones del proceso *Merge*. Pero *M* usa **call** para agregar el próximo valor más chico a la operación de salida *out* apropiada. Además, el proceso *M* usa operaciones semáforo para recibir el próximo valor desde el stream de entrada apropiado. Dentro del módulo, los procesos server implícitos que manejan llamados de *put1* y *put2* son productores; el proceso *M* es consumidor. Sincronizan igual que en el cap. 4.

RENDEZVOUS

Como notamos, RPC por sí mismo provee solo un mecanismo de comunicación intermódulo. Dentro de un módulo, aún necesitamos programar sincronización. Y con frecuencia necesitamos escribir otros procesos para manipular los datos comunicados por medio de RPC.

Rendezvous combina las acciones de servir un llamado con otro procesamiento de la información transferida por el llamado. Con rendezvous, un proceso exporta operaciones que pueden ser llamadas por otros. En esta sección, una declaración de proceso tendrá la siguiente forma:

pname:: declaraciones de operación
declaraciones de variables
sentencias

Las declaraciones de operación tienen la misma forma que en los módulos; especifican los headers de las operaciones servidas por el proceso. También emplearemos arreglos de operaciones, indicadas incluyendo información de rango en la declaración.

Como con RPC, un proceso invoca una operación por medio de una sentencia call, la cual en este caso nombra otro proceso y una operación en ese proceso. Pero en contraste con RPC, una operación es servida por el proceso que la exporta. Por lo tanto, las operaciones son servidas una a la vez en lugar de concurrentemente.

Si un proceso exporta una operación *op*, puede rendezvous con un llamador de *op* ejecutando:

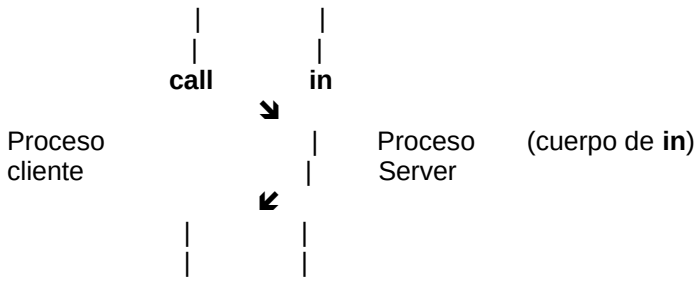
in *op*(formales) → *S* **ni**

Llamaremos a las partes entre las palabras claves *operación guardada*. La guarda nombra una operación y sus parámetros formales; el cuerpo contiene una lista de sentencias *S*. El alcance de los formales es la operación guardada entera.

Esta clase de sentencias de entrada es más poderosa que la sentencia de entrada de SMP. En particular, **in** demora al proceso servidor hasta que haya al menos un llamado pendiente de *op*. Luego elige el llamado pendiente más viejo, copia los argumentos en los formales, ejecuta *S*, y finalmente retorna los parámetros resultado al llamador. En ese punto, ambos procesos (el que ejecuta el **in** y el que llamó a *op*) pueden continuar la ejecución.

El siguiente diagrama muestra la relación entre el proceso llamador y el proceso server que ejecuta **in**:

(9.2)



Como con RPC, el proceso llamador se demora cuando alcanza una sentencia **call**; continúa después de que el proceso server ejecuta la operación llamada. Sin embargo, con rendezvous el server es un proceso activo que ejecuta tanto antes como después de servir una invocación remota. Como indicamos antes, el server también se demorará si alcanza la sentencia **in** y no hay invocaciones pendientes.

La forma básica de **in** sirve una única operación. Como vimos en el cap. 8, la comunicación guardada es muy útil pues permite a un proceso elegir no determinísticamente entre un conjunto de alternativas. Para combinar comunicación guardada con rendezvous, generalizaremos la sentencia **in** como sigue:

in op_1 **and** B_1 **by** $e_1 \rightarrow S_1$
□ .....
□ op_n **and** B_n **by** $e_n \rightarrow S_n$
ni

Cada operación guardada nuevamente nombra una operación y sus formales, y contiene una lista de sentencias. Sin embargo, la guarda también puede contener dos partes opcionales. La segunda parte es una expresión de sincronización (**and** B_i); si se omite, se asume que B_i es true. La tercera parte de una guarda es una expresión de scheduling (**by** e_i); su efecto se describe más adelante. (El lenguaje Ada soporta rendezvous por medio de la sentencia **accept** y comunicación guardada por medio de la sentencia **select**. El **accept** es como la forma básica de **in**, pero el **select** es menos poderoso que la forma general de **in**. Esto es porque **select** no puede referenciar argumentos a operaciones o contener expresiones de scheduling).

Una guarda en una operación guardada tiene éxito cuando (1) la operación fue llamada, y (2) la expresión de sincronización correspondiente es verdadera. Dado que el alcance de los parámetros formales es la operación guardada entera, la expresión de sincronización puede depender de los valores de los formales y por lo tanto de los valores de los argumentos en un llamado. Así, un llamado de una operación podría hacer que la guarda tenga éxito, pero otro llamado de la misma operación podría no hacerlo.

La ejecución de **in** se demora hasta que alguna guarda tenga éxito. Como es usual, si más de una guarda tiene éxito, una de ellas es elegida no determinísticamente. Si no hay expresión de scheduling, la sentencia **in** sirve la invocación más vieja que hace que la guarda tenga éxito. En particular, los argumentos de ese llamado son copiados en los parámetros formales, y luego se ejecuta la lista de sentencias correspondiente. Cuando la lista de sentencias termina, los resultados y el valor de retorno (si los hay) se envían al proceso que llamó la operación. En ese punto, tanto la sentencia **call** como la **in** terminan.

Una expresión de scheduling se usa para alterar el orden de servicio de invocaciones por default (primero la invocación más vieja). En particular, si hay más de una invocación que hace que una guarda en particular tenga éxito, entonces la invocación que minimiza el valor de la expresión de scheduling es servida primero. Como con la expresión de sincronización, una expresión de scheduling puede referirse a los formales en la operación y por lo tanto a los valores reales en un **call**. De hecho, si una expresión de scheduling se refiere solo a variables locales, no tendrá efecto pues su valor no cambiará durante la ejecución de una sentencia de entrada.

Tanto las expresiones de sincronización como las de scheduling son útiles, como veremos. Sin embargo, no están fundamentalmente ligadas al uso de rendezvous. Podrían ser usadas con AMP y SMP. Por ejemplo, podríamos haber permitido que las sentencias **receive** miren sus parámetros.

Ejemplos Cliente/Servidor

Esta sección presenta una serie de ejemplos cliente/servidor que ilustran el uso de sentencias de entrada. Por ahora, asumimos que un programa contiene procesos independientes que no comparten variables. Más adelante combinaremos rendezvous con módulos y RPC.

Consideremos nuevamente el problema de implementar un buffer limitado. En particular, queremos un proceso que exporta dos operaciones: *deposit* y *fetch*. Internamente, el proceso buffer contiene un buffer de hasta *n* ítems de datos. Un proceso productor deposita un ítem en el buffer llamando a la operación *deposit*; un proceso consumidor toma un ítem llamando a la operación *fetch*. Como es usual, *deposit* debe demorarse si ya hay *n* ítems en el buffer, y *fetch* debe demorarse hasta que haya al memos un ítem en el buffer.

El siguiente programa contiene un proceso server que implementa un buffer limitado. Este proceso *Buffer* declara variables locales para representar el buffer, luego repetidamente ejecuta una sentencia de entrada. En cada iteración, *Buffer* espera un llamado por *deposit* o *fetch*. Las expresiones de sincronización en las guardas aseguran que los llamados a *deposit* y *fetch* se demoran cuando es necesario:

```
Buffer:: op deposit(data : T), fetch(var result : T)
        var buf[1:n] : T
        var front := 1, rear := 1, count := 0
        do true →
            in deposit(data) and count < n →
                buf[rear] := data
                rear := rear mod n+1; count := count + 1
            □ fetch(result) and count > 0 →
                result := buf[front]
                front := front mod n+1; count := count - 1
            ni
        od
```

Podríamos comparar este proceso *Buffer* con el monitor del cap. 6. La interfase con los procesos cliente es la misma (y los efectos de llamar a *deposit* y *fetch* también) pero la implementación es diferente. En particular, los cuerpos de procedure en el monitor se convierten en las listas de sentencias en una sentencia de entrada. Además, la sincronización por condición es expresada usando expresiones de sincronización booleanas en lugar de variables condición. Esto es más similar a la forma de programar sincronización por condición usando CCRs.

Como segundo ejemplo, el siguiente programa contiene un proceso server que implementa una solución centralizada al problema de los filósofos. La estructura de *Waiter* es idéntica a la de *Buffer*. En este caso, un llamado a *getforks* puede ser servido cuando ningún vecino está comiendo; un llamado a *relforks* siempre puede ser servido. Un filósofo pasa su índice *i* al *Waiter*, el cual lo usa en la expresión de sincronización en la guarda para *getforks*:

```
Waiter:: op getforks(i : int), relforks(i : int)
        var eating[1:5] := ([5] false)
        { EAT: ( ∀ i: 1 ≤ i ≤ 5: eating[i] ⇒ ¬(eating[iΘ1] ∨ eating[i⊕1]) }
        do true →
            in getforks(i) and not (eating[iΘ1] or eating[i⊕1]) → eating[i] := true
            □ relforks(i) → eating[i] := false
            ni
        od
Phil[i: 1..5]:: do true →
                    call Waiter.getforks(i)
                    come
                    call Waiter.relforks(i)
                    piensa
                od
```

Podríamos comparar este programa con el casi similar usando SMP. La principal diferencia entre los dos programas es que este *Waiter* tiene operaciones *getforks* y *relforks* simples, mientras el *Waiter* del cap. 8 usa arreglos de operaciones. Con SMP, se requerían arreglos de operaciones pues era la única manera en que un filósofo podría identificarse con el waiter. Podríamos haber usado arreglos de operaciones también en esta solución, pero no lo necesitamos pues una sentencia de entrada puede examinar sus parámetros en las expresiones de sincronización. El ser capaz de examinar parámetros provee una herramienta de sincronización poderosa pues podemos sincronizar basado en un valor arbitrario. El próximo ejemplo muestra esto.

El siguiente programa contiene un proceso time server similar en funcionalidad al módulo time server ya definido. El proceso exporta operaciones *get_time* y *delay* para los clientes; también exporta una operación t*ick* que se asume que es llamada por el manejador de interrupciones del reloj. El argumento para *delay* es el tiempo actual en que un proceso cliente quiere ser despertado. Esta es una interfase un poco distinta con el cliente; evita la necesidad de que el server mantenga una cola de procesos durmiendo. Los procesos durmiendo son simplemente aquellos cuyos tiempos de wakeup aún no fueron alcanzados:

```
Time_Server:: op get_time( ) returns time : int
              op delay( waketime : int )
              op tick( )        # llamado por el handler de interrupción del clock
```

```
var tod := 0
do true →
    in get_time( ) returns time → time := tod
    □ delay(waketime) and waketime ≤ tod → skip
    □ tick( ) → tod := tod + 1; reiniciar clock
    ni
od
```

El ejemplo final usa una expresión de scheduling y una expresión de sincronización. El siguiente programa contiene un proceso alocador SJN. La interfase al cliente en este caso es la misma que la del monitor del cap. 6. Pero, como en el proceso time-server, este alocador no necesita mantener colas internas. En lugar de eso solo demora la aceptación de llamados de *request* hasta que el recurso esté libre, y luego acepta el llamado con el menor argumento de *time*:

```
SJN_Allocator:: op request(time : int), release( )
    var free := true
    do true →
        in request(time) and free by time → free := false
        □ release( ) → free := true
        ni
    od
```

Una Red de Ordenación de Merge Filters

Consideremos nuevamente el problema de implementar una red de ordenación usando merge filters, y tratamos de resolverlo usando rendezvous. Hay dos aproximaciones. La primera se asemeja a AMP, la segunda a SMP.

Con AMP, los canales son buffers de mensajes. No hay buffering implícito con rendezvous. Pero podemos programar una red de filtros usando dos clases de procesos: uno para implementar los filtros y uno para implementar buffers de comunicación. En particular, ponemos un proceso buffer (*Buffer*) entre cada par de filtros. Para recibir nueva entrada, un proceso filtro llamaría al buffer entre él y el filtro previo en la red; para enviar una salida, llamaría al buffer entre él y el próximo filtro de la red.

Las redes de filtros son implementadas esencialmente de la misma manera que en UNIX, con los buffers provistos por lo que UNIX llama pipes. Un filtro recibe entrada leyendo de un pipe de entrada (o archivo); envía resultados a un pipe de salida (o archivo). Los pipes no son realmente implementados por procesos (son más como monitores) pero los filtros los usan de la misma manera.

Con la primera aproximación, un proceso filtro hace llamados tanto para recibir entradas como para enviar salidas. También podemos programar filtros usando una segunda aproximación, en la cual un filtro usa una sentencia de entrada para recibir entrada y una sentencia **call** para enviar salida. En este caso, los filtros se comunican directamente uno con otro de manera análoga a usar SMP.

El siguiente programa contiene un arreglo de filtros para merge sorting, programada usando rendezvous. Como en la solución con RPC, cada filtro recibe valores desde operaciones *put1* y *put2* y envía resultados a la operación *out*. También, nuevamente usamos nombrado dinámico para dar a cada proceso, a través de la operación *initial*, una capability para el stream de salida que debería usar. Sin embargo, excepto en estas dos similitudes, los programas son bastante diferentes. Esto es porque rendezvous soporta comunicación directa proceso-a-proceso, mientras RPC no. Esto hace al rendezvous más fácil de usar para programar procesos filtro:

```
optype put(value : int)    # tipo de operaciones para streams de datos
Merge[1:n]:: op put1 : put, put2 : put, initial(c : cap put)
        var in1, in2 : int
        var out : cap put       # capability para stream de salida
        in initial(c) → out := c ni   # toma capability de salida
        in put1(v) → in1 := v ni     # toma dos valores de entrada
```

```
                    in put2(v) → in2 := v ni
                    do in1 ≠ EOS and in2 ≠ EOS →
                        if in1 ≤ in2 → call out(in1)
                                       in put1(v) → in1 := v ni
                        □ in2 ≤ in1 → call out(in2)
                                       in put2(v) → in2 := v ni
                        fi
                    □ in1 ≠ EOS and in2 = EOS →
                       call out(in1)
                       in put1(v) → in1 := v ni
                    □ in1 = EOS and in2 ≠ EOS →
                      call out(in2)
                      in put2(v) → in2 := v ni
                    od
                    call out(EOS)
end
```

Excepto por la sintaxis de las sentencias de comunicación y el uso de nombrado dinámico, este programa es idéntico a lo que podría usarse con SMP. En particular, las sentencias **call** son análogas a los comandos de salida (sentencias !), y las sentencias de entrada son análogas a los comandos de entrada (sentencias ?). Esto es porque las sentencias **call** solo transfieren sus argumentos a otro proceso, y las sentencias de entrada solo asignan los argumentos a variables locales.

Este programa es casi idéntico al merge filter del cap. 7, el cual usa AMP. Nuevamente, las sentencias de comunicación son programadas en forma diferente, pero están exactamente en los mismos lugares. Sin embargo, las sentencias **call** demoran, mientras las sentencias **send** no. Así, la ejecución del proceso es mucho más fuertemente acoplada con rendezvous o SMP que con AMP.

UNA NOTACION DE PRIMITIVAS MULTIPLES

Tanto con RPC como con rendezvous, un proceso inicia la comunicación ejecutando una sentencia **call**, y la ejecución del **call** bloquea al llamador. Esto hace difícil programar algoritmos en los cuales procesos peer intercambian información. Por ejemplo, en un algoritmo heartbeat, procesos vecinos repetidamente intercambian información y luego actualizan su estado local. Esto es fácil de programar con AMP pues podemos hacer que cada proceso primero ejecute **send** y luego **receive**. También podemos programar el intercambio con SMP (aunque no tan fácil y eficientemente) usando comunicación guardada con comandos de entrada y salida en las guardas.

Para programar un intercambio con RPC, un proceso en un módulo debería primero llamar a procedures remotos en módulos vecinos para almacenar su información en los otros módulos. El proceso luego usaría sincronización local para demorarse hasta que la información desde todos los vecinos fue recibida. Esta aproximación indirecta se necesita pues, como notamos antes, los procesos no pueden comunicarse directamente usando RPC.

La situación no es mejor con rendezvous. Aunque los procesos pueden comunicarse directamente, un proceso no puede ejecutar simultáneamente **call** y una sentencia de entrada. Para intercambiar información, procesos vecinos o tienen que ejecutar algoritmos asimétricos o tienen que emplear procesos helper. En el primer caso, tendríamos algunos procesos que primero ejecutan **call** y luego **in** y otros procesos que ejecutan estas sentencias en el orden inverso. En el segundo caso, un proceso regular primero llamaría a los helpers de sus vecinos y luego aceptaría entrada desde su propio helper. (Podríamos haber permitido sentencias **call** en guardas de sentencias de entrada, pero ningún lenguaje lo hizo pues es difícil e ineficiente de implementar).

El resto de esta sección presenta una notación de programación que combina RPC, rendezvous y AMP en un paquete coherente. Esta notación de primitivas múltiples provee un gran poder expresivo pues combina las ventajas de las tres notaciones componentes y provee poder adicional.

Invocación y Servicio de Operaciones

Como con RPC, estructuraremos los programas como colecciones de módulos. Cada módulo tiene la misma forma que antes:

```
module Mname
  declaraciones de operaciones visibles
body
  declaraciones de variables
  código de inicialización
  declaraciones de procedures
  declaraciones de procesos
end
```

Las operaciones visibles son especificadas por declaraciones **op**, que tienen la forma usada antes en este capitulo. Un módulo también puede contener operaciones locales. Nuevamente, emplearemos también arreglos de módulos. Una operación visible puede ser invocada por un proceso o procedure en otro módulo; sin embargo, puede ser servida solo por el módulo que la declara. Una operación local puede ser invocada y servida solo por el módulo que la declara.

Ahora permitiremos que una operación sea invocada por **call** sincrónico o por **send** asincrónico. Las sentencias de invocación tienen las formas:

call *Mname.op*(argumentos)
send *Mname.op*(argumentos)

Como con RPC y rendezvous, una sentencia **call** termina cuando la operación fue servida y los argumentos resultado fueron retornados. Como con AMP, una sentencia **send** termina tan pronto como los argumentos fueron evaluados. Si una operación retorna resultados, puede ser invocada dentro de una expresión; en este caso la palabra clave **call** se omite. (Si una operación tienen parámetros resultado y es invocada por **send**, o si una función es invocada por **send** o no está en una expresión, los valores de retorno son ignorados).

En la notación de primitivas múltiples, también permitiremos que una operación sea servida o por un procedure (**proc**) o por rendezvous (sentencias **in**). La elección la toma el programador del módulo que declara la operación. Depende de si el programador quiere que cada invocación sea servida por un proceso diferente o si es más apropiado rendezvous con un proceso existente. Ejemplos posteriores discuten este tradeoff.

Cuando una operación es servida por un **proc**, un nuevo proceso sirve la invocación. Si la operación fue llamada con **call**, el efecto es el mismo que con RPC. Si la operación fue invocada por **send**, el efecto es la creación de proceso dinámica pues el invocador sigue asincrónicamente con el proceso que sirve la invocación. En ambos casos nunca hay una cola de invocaciones pendientes pues cada una puede ser servida inmediatamente.

La otra manera de servir operaciones es usando sentencias de entrada, las cuales tienen la forma dada en la sección de rendezvous. En este caso, hay una cola de invocaciones pendientes asociada con cada operación; el acceso a la cola es atómico. Una operación es seleccionada para ser servida de acuerdo a la semántica de las sentencias de entrada. Si tal operación es llamada, el efecto es rendezvous pues el llamador se demora. Si tal operación es invocada por **send**, el efecto es similar a AMP pues el emisor continúa.

En resumen, hay dos maneras de invocar una operación (**call** y **send**) y dos maneras de servir una invocación (**proc** e **in**). Estas 4 combinaciones tienen los siguientes efectos:

(9.11)

invocación	*servicio*	*efecto*
call	**proc**	llamado a procedure
call	**in**	rendezvous
send	**proc**	creación de procesos dinámica
send	**in**	AMP

Un llamado a procedure será local si el llamador y el **proc** están en el mismo módulo; en otro caso, será remoto. Una operación no puede ser servida tanto por un **proc** como por sentencias de entrada pues el significado no sería claro (La operación es servida inmediatamente o encolada?). Sin embargo, una operación puede ser servida por más de una sentencia de entrada, y estas pueden estar en más de un proceso en el módulo que declara la operación. En este caso, los procesos comparten la cola de invocaciones pendientes (y la acceden atómicamente).

Con monitores y AMP, definimos una primitiva **empty** para determinar si una variable condición o canal de mensajes estaba vacío. En este capítulo usaremos una primitiva similar pero con alguna diferencia. En particular, si *op* es un nombre de operación, entonces ?*op* es una función que retorna el número de invocaciones pendientes de *op* (no debe confundirse con el comando de entrada de SMP). Esto es con frecuencia útil en sentencias de entrada. Por ejemplo, lo siguiente da prioridad a *op1* sobre *op2*:

in *op1*(...) → *S1* □ *op2*(...) **and** ?*op1* = 0 → *S2* **ni**

La expresión de sincronización en la segunda operación guardada permite que *op2* sea seleccionada solo si no hay invocaciones de *op1* en el momento en que ?*op1* es evaluada.

Ejemplos

Tres pequeños ejemplos, muy relacionados, ilustrarán las maneras en que las operaciones pueden ser invocadas y servidas. Primero consideramos la implementación de una cola secuencial:

```
module Queue
  op deposit(item : T), fetch(res item : T)
body
  var buf[1:n] : T, front := 1, rear := 1, count := 0
  proc deposit(item)
    if count < n → buf[rear] := item
                   rear := rear mod n+1; count := count + 1
    □ count = n → tomar acciones apropiadas para overflow
    fi
  end
  proc fetch(item)
    if count > 0 → item := buf[front]
                   front := front mod n+1; count := count - 1
    □ count = 0 → tomar acciones apropiadas para underflow
    fi
  end
end
```

Cuando es invocada *deposit*, un nuevo item es almacenado en *buf*. Si *deposit* es llamada, el invocador espera; si es invocada por **send**, el invocador continúa antes de que el item sea realmente almacenado (en cuyo caso el invocador debería estar seguro de que no ocurrirá overflow. Cuando *fetch* es invocado, un ítem es removido de *buf*; en este caso, *fetch* necesita ser invocada por una sentencia **call**, o el invocador no recibirá el resultado.

El módulo *Queue* es adecuado para usar por un único proceso en otro módulo. No puede ser compartido por más de un proceso pues la implementación no emplea ninguna sincronización. En particular, podría resultar interferencia si las operaciones son invocadas concurrentemente. Si necesitamos una “cola sincronizada”, podemos cambiar el módulo *Queue* por uno que implemente un buffer limitado.

El siguiente programa da un módulo de buffer limitado:

```
module BoundedBuffer
  op deposit(item : T), fetch(res item : T)
body
```

```
    var buf[1:n] : T, front := 1, rear := 1, count := 0
    Buffer:: do true →
                in deposit(item) and count < n →
                    buf[rear] := item; rear := rear mod n+1; count := count + 1
                □ fetch(item) and count > 0 →
                    item := buf[front]; front := front mod n+1; count := count - 1
                ni
            od
end
```

Las operaciones visibles son las mismas que en *Queue*. Sin embargo, las operaciones son servidas por una sentencia de entrada en un único proceso. Así, las invocaciones son servidas una a la vez. Además, las expresiones de sincronización se usan para demorar el servicio de *deposit* hasta que haya lugar en *buf* y para demorar el servicio de *fetch* hasta que *buf* contenga un item. Nuevamente, *fetch* debería ser invocada por una sentencia **call**, pero ahora es perfectamente seguro invocar a *deposit* por una sentencia **send** (a memos que haya tantas invocaciones pendientes de *deposit* que todos los kernel buffers estén usados). Esto es porque *deposit* no retorna resultados.

Los módulos en estos ejemplos ilustran dos maneras distintas de implementar la misma interfase. La elección la tiene el programador y depende de cómo será usada la cola. De hecho, hay otra manera de implementar un buffer limitado que ilustra otra combinación de las distintas maneras de invocar y servir operaciones en la notación de primitivas múltiples.

Consideremos la siguiente sentencia de entrada, la cual espera una invocación, luego asigna los parámetros a variables en el proceso o módulo que ejecuta la sentencia:

in *op*(f1, ..., fn) → v1 := f1; ...; vn := fn **ni**

Esta forma de **in** es idéntica a la sentencia **receive** del cap. 7:

receive *op*(v1, ..., vn)

Dado que **receive** es solo una abreviación para un caso especial de **in**, usaremos **receive** cuando así es como queremos servir una invocación.

Ahora consideremos una operación que no tiene argumentos y que es invocada por **send** y servida por **receive** (o la sentencia **in** equivalente). En este caso, la operación es equivalente a un semáforo, con **send** siendo un **V**, y **receive** siendo un **P**. El valor inicial del semáforo es cero; su valor corriente es el número de mensajes “null” que fueron enviados a la operación, menos el número que fue recibido.

El siguiente programa da una segunda implementación de *BoundedBuffer* que usa semáforos para sincronización. Las operaciones *deposit* y *fetch* son servidas por procedures como en *Queue*. Por lo tanto, podría haber más de una instancia de estos procedures activas a la vez. Sin embargo, aquí usamos operaciones semáforo para implementar la sincronización requerida para un buffer limitado. Los semáforos implementan exclusión mutua y sincronización por condición. La estructura de este módulo es como la de un monitor, pero la sincronización es implementada usando semáforos en lugar de exclusión implícita y variables condición explícitas:

```
module BoundedBuffer
  op deposit(item : T), fetch(res item : T)
body
  var buf[1:n] : T, front := 1, rear := 1, count := 0
  # operaciones locales usadas para simular semáforos
  op empty( ), full( ), mutexD( ), mutexF( )
  send mutexD( ); send mutexF( )   # inicializa los semáforos
  fa i := 1 to n → send empty( ) fa
  proc deposit(item)
    receive empty( ); receive mutexD( )
    buf[rear] := item; rear := rear mod n+1
    send mutexD( ); send full( )
```

```
    end
    proc fetch(item)
      receive full( ); receive mutexF( )
      item := buf[front]; front := front mod n+1
      send mutexF( ); send empty( )
    end
  end
```

Las dos implementaciones de un buffer limitado ilustran un paralelo importante entre expresiones de sincronización en sentencias de entrada y sincronización explícita en procedures. Con frecuencia son usados para los mismos propósitos. También, dado que las expresiones de sincronización en sentencias de entrada pueden depender de argumentos de invocaciones pendientes, las dos técnicas de sincronización son igualmente poderosas. Sin embargo, a menos que uno necesite la concurrencia provista por múltiples invocaciones de procedures, es más eficiente que los clientes rendezvous con un único proceso en lugar de que cada uno de sus procesos server sincronice con cada uno de los otros. Esto es porque la creación de procesos toma tiempo, y los procesos consumen más espacio que las colas de mensajes.

CLIENTES Y SERVIDORES

Con la notación de primitivas múltiples, podemos usar AMP para programar filtros casi exactamente como en el cap. 7. La principal diferencia es “cosmética”: los procesos son encapsulados por módulos. Dado que la notación incluye tanto RPC como rendezvous, también podemos programar clientes y servidores como en las secciones correspondientes de este capítulo. Pero la nueva notación provee flexibilidad adicional, la cual ilustraremos. Primero desarrollamos otra solución al problema de lectores/escritores; a diferencia de las previas, esta solución encapsula el acceso a la BD. Luego se da una implementación de archivos replicados, los cuales se discutieron en el cap. 7. Finalmente, se describen dos maneras adicionales para programar un algoritmo probe/echo para resolver el problema de la topología de red.

Lectores/Escritores Revisitado: Acceso Encapsulado

Recordemos que en este problema, dos clases de procesos comparten una BD. Los lectores examinan la BD y pueden hacerlo concurrentemente; los lectores modifican la BD, de modo que necesitan acceso exclusivo. En soluciones anteriores (como en monitores) los procesos tenían que seguir el protocolo de pedir permiso antes de acceder la BD, luego liberar el control cuando terminaban. Un monitor no puede encapsular el acceso en sí mismo pues solo un proceso por vez puede ejecutar en un monitor, y entonces los lectores no podrían ejecutar concurrentemente.

El siguiente programa presenta una solución en la cual un único módulo encapsula el acceso a la BD. Los procesos clientes simplemente llaman a la operación *read* o *write*; el módulo oculta cómo se sincronizan estos llamados:

```
module ReadersWriters
  op read(res results), write(nuevos valores)
body
  op startread( ), endread( )
  var almacenamiento para la BD o buffers de transferencia de archivos
  proc read(results)
    call startread( )
    lee la BD
    send endread( )
  end
  Writer:: var nr := 0
           do true →
                in startread( ) → nr := nr + 1
                □ endread( ) → nr := nr - 1
                □ write(valores) and nr = 0 → escribe la BD
```

```
        ni
    od
```

El módulo usa tanto RPC como rendezvous. La operación *read* es implementada por un **proc**, de modo que múltiples lectores pueden ejecutar concurrentemente. Por otro lado, la operación *write* es implementada por rendezvous con el proceso *Writer*; esto asegura que las escrituras son servidas una a la vez. El módulo también contiene dos operaciones locales, *startread* y *endread*, los cuales son usados para asegurar que *read* y *write* son mutuamente exclusivos. El proceso *Writer* también sirve a éstas, de modo que puede mantener la pista del número de lectores activos y demorar a *write* cuando es necesario.

Esta solución da prioridad a los lectores. Usando la función ?, podemos cambiar la sentencia de entrada como sigue para dar prioridad a los escritores:

```
in startread( ) and ?write = 0 → nr := nr + 1
 endread( ) → nr := nr - 1
 write(valores) and nr = 0 → escribe la BD
ni
```

Esto demora el servicio de invocaciones de *startread* siempre que haya invocaciones pendientes de *write*.

Este programa “lockea” la BD entera para los lectores o para un escritor. Esto es satisfactorio normalmente para pequeños archivos de datos. Sin embargo, las transacciones de BD típicamente necesitan lockear registros individuales. Esto es porque una transacción no sabe en principio cuales registros accederá; por ejemplo, tiene que examinar algún registro antes de saber qué examinar a continuación. Para implementar esta clase de lockeo dinámico, podríamos emplear módulos múltiples, uno por registro de la BD. Sin embargo, esto no encapsularía el acceso a la BD. Alternativamente, podríamos emplear un esquema de lockeo más elaborado y finer-grained dentro del módulo *ReadersWriters*. Por ejemplo, cada *read* y *write* podría adquirir los locks que necesita. Los DBMS típicamente usan esta aproximación.

Archivos replicados

En el cap. 7 describimos el problema de implementar archivos replicados. Asumimos nuevamente que hay *n* copias de un archivo y que deben mantenerse consistentes. Ya describimos formas de usar tokens para asegurar la consistencia. Aquí se presenta una solución que usa locking.

Supongamos que hay *n* módulos server y cualquier número de procesos clientes. Cada módulo server provee acceso a una copia del archivo. Cada cliente interactúa con uno de los módulos server, por ejemplo, uno que ejecuta en el mismo procesador que el cliente.

Cuando un cliente quiere acceder al archivo, llama a la operación *open* de su server e indica si el archivo será leído o escrito. Luego de abrir un archivo, un cliente accede al archivo llamando a las operaciones *read* o *write* de su server. Si el archivo fue abierto para lectura, solo se puede usar *read*; en otro caso tanto *read* como *write* pueden usarse. Cuando un cliente termina de acceder al archivo, llama a la operación *close* de su server.

Los file servers interactúan para asegurar que las copias del archivo se mantienen consistentes y que a lo sumo a un cliente a la vez se le permite escribir en el archivo. Cada instancia de *RepFile* tiene un proceso manejador local que implementa una solución al problema de lectores/escritores. Cuando un cliente abre un archivo para lectura, la operación *open* llama a la operación *startread* del manager de lock local. Sin embargo, cuando un cliente abre un archivo para escritura, la operación *open* llama a la operación *startwrite* de los *n* managers de lock.

El siguiente programa contiene el módulo de archivo replicado:

```
module RepFile[myid: 1..n]
  type mode = (READ, WRITE)
  op open(m : mode), close( ),          # operaciones de cliente
```

```
        read(res results), write(values)
    op remote_write(values), startwrite( ), endwrite( )     # operaciones de server
body
    op startread( ), endread( )    # operaciones locales
    var file buffers, use : mode
    proc open(m)
      if m = READ → call startread( )     # toma el lock de read local
                    use := READ
      □ m = WRITE → # toma los write locks de todas las copias del archivo
                    fa i := 1 to n → call RepFile[i].startwrite( ) af
                    use := WRITE
      fi
    end
    proc close( )
      if use = READ → # libera el lock de read local
                      send endread( )
      □ use = WRITE → # libera todos los write locks
                      co i := 1 to n → send RepFile[i].endwrite( ) oc
      fi
    end
    proc read(results)
      lee de la copia local del archivo y retorna resultados
    end
    proc write(values)
      if use = READ → error: archivo no abierto para escritura fi
      escribe valores en la copia local del archivo
      # actualiza concurrentemente todas las copias remotas
      co i := 1 to n st i ≠ myid → call RepFile[i].remote_write(values) oc
    end
    proc remote_write(values)       # llamado por otros file servers
      escribe valores en la copia local del archivo
    end
    Lock:: var nr := 0, nw := 0
          do true →
              in startread( ) and nw = 0 → nr := nr + 1
              □ endread( ) → nr := nr - 1
              □ startwrite( ) and nr = 0 → nw := nw + 1
              □ endwrite( ) → nw := nw - 1
              ni
          od
end RepFile
```

Hay varios aspectos interesantes de esta implementación:

* Cada módulo *RepFile* exporta dos conjuntos de operaciones: unas llamadas por sus clientes y otras llamadas por otros módulos *RepFile*. Las operaciones *open*, *close*, y de acceso son implementadas por procedures, aunque solo *read* tiene que permitir concurrencia. Las operaciones de locking son implementadas por rendezvous con el manager de lock.

* Cada módulo mantiene la pista del modo de acceso corriente, es decir, la última manera en que fue abierto el archivo. Usa *use* para asegurar que el archivo no es escrito si fue abierto para lectura y para determinar qué acciones tomar cuando el archivo es cerrado. Sin embargo, un módulo no se protege a sí mismo de un cliente que accede un archivo sin primero abrirlo. Esto puede ser resuelto usando nombrado dinámico o autenticación.

* Dentro del procedure *write*, un módulo primero actualiza su copia local del archivo y luego actualiza concurrentemente cada copia remota. Una aproximación alternativa es actualizar solo la copia local en *write* y luego actualizar todas las copias remotas cuando el archivo es cerrado. Eso es análogo a usar una política de cache write-back como opuesta a una política write-through.

* Los clientes adquieren permiso para escribir desde los procesos lock uno a la vez y en el mismo orden. Esto evita el deadlock que podría resultar si dos clientes adquirieron permiso en ordenes distintos. Sin embargo, los clientes liberan el permiso de escritura concurrentemente y usan **send** más que **call** pues no hay necesidad de demora en este caso.

* El proceso manager de lock implementa la solución clásica al problema de lectores/escritores. Es un monitor activo.

ALGORITMOS PARALELOS

En esta sección, se desarrollan algoritmos paralelos para dos nuevos problemas. el primero es labeling de regiones, el cual se da en procesamiento de imágenes. Resolvemos el problema usando un algoritmo heartbeat. El segundo ejemplo es el problema del viajante. Lo resolvemos usando procesos worker replicados. Las soluciones usan la notación de primitivas múltiples.

Region Labeling: Algoritmo Heartbeat

Una imagen consta de una matriz de pixels. Cada pixel tiene un valor que representa su intensidad de luz. un problema común del procesamiento de imágenes es identificar regiones consistentes de pixels vecinos que tienen el mismo valor. Esto se llama problema de region labeling pues el objetivo es identificar y asignar un único label a cada región.

Para hacer el problema más preciso, asumimos que los valores de pixels son almacenados en la matriz *image*[1:n, 1:n]. Los vecinos de un pixel son los 4 pixels a su alrededor. dos pixels pertenecen a la misma región si son vecinos y tienen el mismo valor. Así, una región es un conjunto máximo de pixels que están conectados y tienen el mismo valor.

El label de cada pixel se almacena en una segunda matriz *label*[1:n, 1:n]. Inicialmente, a cada punto se le da un único label, por ejemplo, sus coordenadas $n * i + j$. El valor final de *label*[i,j] es el mayor de los labels iniciales en la región a la cual pertenece el pixel [i,j].

La manera natural de resolver este problema es por medio de un algoritmo iterativo. En cada iteración, examinamos cada pixel y sus vecinos. Si el pixel y un vecino tienen el mismo valor, entonces seteamos su label al máximo de su label corriente y el de su vecino. Podemos hacer esto en paralelo para cada pixel, asumiendo que almacenamos los nuevos labels en una matriz separada.

El algoritmo termina si ningún label cambia durante una iteración. Si las regiones son compactas fairly, el algoritmo terminará después de aproximadamente $O(n)$ iteraciones. Sin embargo, en el peor caso, se requieren $O(n^2)$ iteraciones pues podría haber una región que “serpentea” por la imagen.

Aunque podrían usarse n^2 tareas paralelas en este problema, cada una es demasiado chica para ser eficiente. Más aún, n normalmente es grande (por ej, 1024), de modo que el número de tareas será mucho más grande que el número de procesadores disponibles. Supongamos que tenemos m^2 procesadores y que n es un múltiplo de m. Entonces una buena manera de resolver el problema de region labeling es particionar la imagen en m^2 bloques y emplear un proceso para cada bloque.

Cada proceso computa los labels para los pixels de su bloque. Si *image* y *label* están almacenadas en memoria compartida, entonces cada proceso puede ejecutar un algoritmo iterativo paralelo y usar sincronización barrier al final de cada iteración, como en el cap. 3. Sin embargo, aquí asumiremos que los procesos no comparten variables. Por lo tanto, *image* y *label* necesitan ser distribuidas entre los procesos.

Dado que las regiones pueden sobrepasar los límites de bloques, cada proceso necesita interactuar con sus vecinos en cada iteración. Esto lleva a un algoritmo heartbeat. En particular, en cada iteración un proceso intercambia los labels de pixels sobre el límite de su bloque con sus 4 procesos vecinos. Luego computa nuevos labels.

El siguiente programa contiene un outline de los módulos que manejan cada bloque. Cada módulo calcula labels para su porción de la imagen. Cada uno también almacena los valores y labels de los pixels en los límites de bloques vecinos. Los módulos vecinos usan las operaciones *north*, *south*, *east* y *west* para intercambiar labels de límites. No se muestran detalles de cómo *Block* usa estas operaciones y computan nuevos labels:

```
module Block[i: 1..m, j: 1..m]
  const p := n/m
  op north(border[1:p] : int), south(border[1:p] : int)
  op east(border[1:p] : int), west(border[1:p] : int)
  op answer(change : int)
body
  B:: var image[0:p+1, 0:p+1] : int   # inicializado con los valores de pixels
      var label[0:p+1, 0:p+1] : int     # inicializado con labels únicos
      var change := true
      do change →
              intercambia labels de bordes con vecinos
              change := false
              actualizar label, setear change a true si hay cambios de label
              send Coordinator.result(change)
              receive answer(change)
      od
end
```

El módulo *Block* también exporta una operación, *answer*, que se usa para determinar cuando terminar. En el algoritmo heartbeat para el problema de la topología de red en el cap. 7, un proceso podía terminar cuando no recibía nueva información de topología en una iteración. Aquí, sin embargo, un proceso no puede determinar por sí mismo cuando terminar. Aún si no hay cambio local en una iteración, podría haber cambiado un label en otro bloque, y ese pixel podría pertenecer a una región que cubre más de un bloque.

Una manera de detectar terminación global es hacer que cada bloque ejecute el número de operaciones del peor caso. Sin embargo, para este problema, esto sería mucho más de lo que normalmente se requiere. En consecuencia, usamos un módulo coordinador para detectar terminación. Este módulo se muestra a continuación. Al final de cada iteración, cada bloque le dice al coordinador si alguno de sus labels cambió el valor. El coordinador recolecta estos mensajes y luego informa a cada bloque si hubo algún cambio. Si es así, los bloques ejecutan otra iteración; si no, terminan:

```
module Coordinator
  op result(localchange : bool)
body
  Check:: var change : bool := true
          do change →
                change := false
                # mira si hubo cambios en algún bloque
                fa i := 1 to m, j := 1 to m →
                      in result(ch) → change := change or ch ni
                af
                # broadcast resultado a cada bloque
                co i := 1 to m, j := 1 to m →
```

```
                    send Block[i,j].answer(change)
                oc
            od
        end
```

Usar un coordinador central agrega $2m^2$ mensajes de overhead en cada ronda. Como está programado, el coordinador recolecta todos los mensajes *result*, y lo hace uno por vez. Aunque estos varios mensajes tienen que ser recolectados, podríamos usar múltiples procesos coordinadores para recogerlos en paralelo. Por ejemplo, podríamos usar *m* coordinadores (uno por fila por ejemplo) para recoger resultados de cada fila, y luego usar un coordinador más para recoger los resultados desde los otros coordinadores.

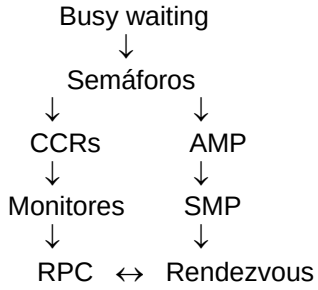
La computación principal en este programa usa AMP para comunicarse con el coordinador. Si el coordinador pudiera responder a invocaciones en un orden distinto en que las recibió, los procesos en el cómputo principal podrían ejecutar simplemente:

```
change := Coordinator.result(change)
```

No tendrían que ejecutar **send** y luego **receive**. Algunos lenguajes soportan esta funcionalidad. Por ejemplo, SR tiene una sentencia **forward**, la cual toma una invocación que está siendo servida y la forward a otra operación sin hacer reply de ella nuevamente. El llamador se desbloquea cuando la invocación de la segunda operación es servida (a menos que también use **forward**).

Práctica

Los capítulos anteriores describieron e ilustraron los mecanismos más comunes para sincronización entre procesos. Podemos verlos e ilustrar las relaciones entre ellos de la siguiente manera:



Por ejemplo, los monitores emplean la señalización explícita de los semáforos y la exclusión mutua implícita de CCRs, AMP extiende los semáforos con datos, y tanto RPC como rendezvous combinan la interfase procedural de los monitores con SMP. Como se ve, los mecanismo aparecen formando un círculo cerrado. No ha habido ninguna propuesta nueva fundamental en una década. Sin embargo, hubo variaciones interesantes, como se describirán.

Las técnicas de sincronización que se describieron caían en dos grandes categorías: las que usan variables compartidas y las que usan MP. Corresponden a las dos mayores categorías de multiprocesadores: aquella en la cual los procesadores pueden acceder memoria compartida y la que procesadores y memorias independientes están conectados por una red de comunicación. Como vimos, un programa escrito usando MP puede ser ejecutado en una máquina de memoria compartida. También hay sistemas de software que proveen la ilusión de una memoria virtual compartida y por lo tanto permiten que programas de variables compartidas sean ejecutados en máquinas de memoria distribuida.

Durante muchos años, la programación concurrente fue practicada solo por programadores de sistemas que construían sistemas operativos o sistemas de BD. Pero con el incremento de arquitecturas multiprocesador, muchos programadores de aplicaciones ahora tienen la necesidad (y oportunidad) de escribir programas concurrentes. Esto es esencialmente verdadero en la comunidad científica pues el paralelismo es la única manera de solucionar la limitación fundamental que la velocidad de la luz impone sobre la performance.

Para facilitar la escritura de programas concurrentes, los fabricantes de workstations científicas y de multiprocesadores proveen rutinas que soportan creación de procesos y un conjunto de los mecanismos de sincronización que describimos.

Uno puede escribir un programa concurrente para una máquina específica comenzando con un programa secuencial y luego usando una librería de subrutinas para creación de procesos, sincronización y comunicación. Esta es la forma en que están escritas varias aplicaciones paralelas corrientes. Sin embargo, esta aproximación tiende a programar en lenguaje ensamblador pues el programador tiene que tratar con la arquitectura de la máquina explícitamente, empaquetar y desempaquetar mensajes, hacer chequeos de tipo intermáquina, etc. Más aún, el programa resultante no es portable pues usa la librería de una máquina y SO específico.

Una aproximación mucho más atractiva es emplear un lenguaje de programación concurrente de alto nivel. El programador entonces puede concentrarse en la lógica de la aplicación y dejar a los implementadores del lenguaje que se preocupen por cómo emplear las rutinas de bajo nivel de una máquina especial. Fueron diseñados numerosos lenguajes de programación concurrentes, y muchos fueron implementados. Daremos una sinopsis de los más importantes. También se discuten los tradeoffs entre ellos y su performance relativa.

Lenguajes

Desde hace mucho tiempo hubo lenguajes de programación con mecanismos para especificar concurrencia, comunicación y sincronización, por ej, PL/I y Algol 68. Sin embargo, estos viejos lenguajes estaban pensados principalmente para programación secuencial, y los mecanismo de concurrencia fueron “agregados”. Recién en las dos últimas décadas se vieron lenguajes especiales para programación concurrente y en los cuales las características de concurrencia juegan un rol central.

Dado que hay varias clases de mecanismos de programación concurrente hay numerosos lenguajes de programación concurrente. Los siguientes son algunos de los más conocidos e importantes, y en ellos se incluyen los mecanismos de sincronización que ya vimos (uno o más):

Lenguaje	*Mecanismo(s) de comunicación*
Actors	AMP (orientado a objetos)
Ada	Rendezvous
Argus	RPC + CCRs + transacciones atómicas
Concurrent C	Rendezvous + send asincrónico
Concurrent Euclid	Monitores (disciplina SW)
Concurrent Pascal	Monitores (disciplina SX)
CONIC	AMP
CSP	SMP
DP	RPC + CCRs
Edison	CCRs
Emerald	RPC + monitores (orientado a objetos)
Gypsy	AMP
Joyce	SMP
Linda	mensajes con espacio de tuplas compartido
Lynx	RPC + rendezvous + CCRs
Mesa	RPC + monitores (disciplina SC)
Modula	monitores (disciplina SW)
Modula-3	package con corrutinas + locks + monitores (SC)
NIL	AMP + rendezvous
Occam	SMP
Pascal Plus	monitores (disciplina SU)
Path Pascal	módulos + path expressions
PLITS	AMP
SR	multiple shared and message primitives
StarMod	multiple message primitives
Turing Plus	monitores (disciplinas SC y SW)

Hay también una gran variación en la componente secuencial de lenguajes de programación concurrentes. Algunos extienden un lenguaje secuencial (como C o Pascal) mientras otros son diseñados especialmente (aunque ciertamente influenciados por otros lenguajes). Basar un lenguaje concurrente en un lenguaje secuencial tiene el beneficio de mayor familiaridad. Pero con frecuencia es posible proveer una integración más limpia de las características secuenciales y concurrentes de un lenguaje diseñándolas juntas.

Ningún lenguaje de programación concurrente es dominante. Esto se debe a la variedad de los modelos de programación, las arquitecturas de hardware, y las aplicaciones; también debido a lo novedoso del campo. En este capítulo se da una sinopsis de algunos de los lenguajes más importantes.

* Turing Plus: el lenguaje basado en monitores más reciente
* Occam: derivación de CSP y lenguaje para los transputers
* Ada: el lenguaje del US Department of Defense para embedded systems
* SR: lenguaje con múltiples primitivas que soporta tanto memoria compartida como programación distribuida

* Linda: colección de 4 primitivas básicas que pueden agregarse a cualquier lenguaje de programación secuencial

Cada uno de estos lenguajes contiene características novedosas, está en uso y disponible en varias máquinas. Colectivamente ilustran el rango de posibilidades para comunicación y sincronización así como para modularización, nombrado de componentes, y ubicación de componentes (es decir, mapeo a hardware).

También se comparan estos 5 lenguajes y se discuten algunos temas de performance en programación concurrente, incluyendo la eficiencia relativa de distintos mecanismos de comunicación y sincronización.

TURING PLUS: MONITORES

Hay varios lenguajes de programación concurrente basados en monitores. De hecho, cada una de las 5 disciplinas de señalización en monitores fueron incorporadas en al menos un lenguaje. Esta sección describe Turing Plus, que es el más reciente de tales lenguajes y es bastante representativo de los otros.

Turing es un lenguaje secuencial desarrollado en 1982-1983 para computación instruccional. El objetivo de los diseñadores fue crear un lenguaje que sea fácil de aprender y usar, expresivo, y eficiente. Además, querían una definición formal precisa matemáticamente. Turing Plus fue diseñado en la mitad de los ‘80s para soportar un rango mayor de aplicaciones, incluyendo programación de sistemas. Turing Plus extiende Turing con procesos, monitores y variables compartidas y con características adicionales como compilación separada, manejo de excepciones, y monitores controladores de dispositivos.

Estructura de programa

Un programa Turing consta de una única unidad de compilación. A diferencia de muchos otros lenguajes, no hay “packaging” alrededor del programa; simplemente comienza con una declaración o sentencia. Por ejemplo, lo siguiente es un programa completo:

```
put “Hello world”
```

En Turing Plus, un programa se compone de declaraciones, sentencias, subprogramas (procedimientos y funciones), procesos, módulos y monitores. Un programa Turing Plus puede ser compilado como una sola unidad. Alternativamente, subprogramas, módulos y monitores pueden ser movidos fuera del programa principal y compilados separadamente. En este caso, cada una de tales componentes del programa principal es reemplazada por una directiva child, la cual nombra el archivo en el cual se ubicó la componente. A su vez, la componente child comienza con una directiva parent, la cual nombra el archivo que contiene la componente principal. También soporta mayor anidamiento de componentes parent/child.

Un módulo que es compilado como una única unidad tiene una forma similar a la introducida en el cap. 9:

```
module name
  import (identificadores globales)
  export (identificadores locales)
  declaraciones y sentencias
end name
```

La cláusula import hace que las declaraciones globales sean accesibles al módulo. Pueden incluir variables, subprogramas, tipos, etc. La cláusula export hace que objetos declarados localmente sean accesibles fuera del módulo; pueden incluir subprogramas, constantes y tipos. Sin embargo, los módulos no pueden exportar variables; esto asegura que los detalles de representación se ocultan.

Cuando un módulo es compilado separadamente, se separa en un stub y un cuerpo. El stub contiene las cláusulas import y export y las declaraciones de objetos exportados; el stub contiene solo los headers de los procedures o funciones exportados. El cuerpo de un módulo contiene los detalles de implementación, por ejemplo, variables locales y los cuerpos de los subprogramas.

Los monitores son una clase especial de módulo. Pueden ser compilados o como una sola unidad o separadamente con un stub y un cuerpo.

La declaración de proceso en Turing Plus tiene la forma básica:

```
process name(formales)
   declaraciones y sentencias
end name
```

Los procesos pueden ser declarados en la componente principal y en los módulos. Sin embargo, no pueden ser declarados dentro de monitores pues conflictuarían con la exclusión mutua asumida dentro de los monitores. Las declaraciones de procesos son patrones, como las declaraciones de procedures. Una instancia de un proceso se crea por medio de la sentencia **fork**:

```
fork name(reales)
```

Esto crea una instancia del proceso *name* y le pasa los parámetros reales. Los nuevos procesos luego ejecutan concurrentemente con el proceso que ejecuta **fork**.

Turing Plus contiene las clases usuales de declaraciones y sentencias. La sintaxis es básicamente Pascal. Sin embargo, el lenguaje contiene 4 características adicionales que están relacionadas con la verificación de programa: **assert** B, **pre** B, **post** B, e **invariant** B. En cada caso, B es una expresión booleana. El programador puede usar estas características para especificar aserciones críticas. Más aún, si se habilita el chequeo en run-time, las aserciones son chequeadas durante la ejecución; si alguna aserción es falsa, la ejecución del programa aborta.

La sentencia **assert** puede ponerse en los lugares donde es posible poner una sentencia normal. Las cláusulas **pre** y **post** pueden ponerse al comienzo de un subprograma, módulo o monitor. En un subprograma, **pre** especifica una precondición que debe ser true después de cada llamado; **post** especifica una postcondición que debe ser true antes de cada retorno. En un header de módulo o monitor, **pre** especifica qué debe ser true al comienzo de la inicialización, y **post** especifica qué debe ser true al final de la inicialización. La cláusula invariante puede ponerse al comienzo de un módulo o monitor, en cuyo caso especifica el invariante del módulo o el monitor. También puede ser agregada a la sentencia **loop** que especifica un loop invariant.

Interacción de procesos

Los procesos en los programas Turing Plus interactúan usando variables globales, monitores o ambos. Por esto, pueden ser ejecutados en monoprocesadores (usando multiplexing) o en multiprocesadores de memoria compartida.

Como dijimos, los monitores son clases especiales de módulos. Como es usual, los procesos en los monitores ejecutan con exclusión mutua y usan variables condición para sincronización por condición. Turing Plus provee 4 clases de variables condición: regular, diferida, prioridad y timeout. Las variables condición son accedidas usando sentencias **wait** y **signal** y una función **empty**.

Las variables condición regulares son colas FIFO de procesos en espera con semántica signal-and-wait (SW); es decir, el **signal** es preemptivo. Las variables condición diferidas son colas FIFO de procesos en espera con semántica signal-and-continue (SC). Las variables condición prioridad proveen wait con prioridad más señalización SW. Finalmente, las variables condición timeout soportan interval timing. Un proceso especifica un intervalo de delay en la sentencia **wait**; es despertado por un signal implícito, no preemptivo cuando expiró el intervalo.

Para programación de sistemas de bajo nivel, Turing Plus provee una clase especial de monitor llamado "device monitor". Como sugiere su nombre, un device monitor se usa para programar la interfase a un dispositivo de hardware. Turing Plus implementa exclusión mutua en device monitors inhibiendo interrupciones de hardware. Tales monitores también pueden contener procedures de manejo de interrupción, los cuales son llamados cuando ocurren las interrupciones de hardware. Otros mecanismos de bajo nivel permiten que un programa Turing Plus acceda registros del dispositivo.

Todas las variables condición en device monitors debe ser diferidas. Esto asegura que los procedures de manejo de interrupción no son preemptados cuando señalan procesos regulares. Esto hace eficientes a los manejadores de interrupción. También hace mucho más fácil implementar exclusión mutua por medio de la inhibición de interrupciones.

OCCAM: PASAJE DE MENSAJES SINCRONICO

Occam es un lenguaje distintivo, más que cualquier otro de los descriptos en este capítulo. Contiene un número muy chico de mecanismos, tiene una sintaxis única, las sentencias básicas (asignación y comunicación) son en sí mismas vistas como procesos primitivos, y un programa Occam contiene un número estático de procesos y paths de comunicación estáticos.

La notación de programación CSP introdujo SMP; sin embargo, CSP en sí mismo nunca fue implementado. Occam es el derivado del CSP más usado y conocido. La primera versión de Occam fue diseñada en la mitad de los 80s. Aquí se describe la última versión, Occam 2, aparecida en 1987. Aunque Occam es un lenguaje por derecho propio, siempre estuvo asociado con el *transputer* (multiprocesador de memoria distribuida de bajo costo implementado en VLSI). De hecho, Occam es esencialmente el lenguaje de máquina del transputer.

Estructura de programa

Las unidades básicas de un programa Occam son declaraciones y tres "procesos" primitivos: asignación, input y output. Un proceso asignación es simplemente una sentencia de asignación. Los procesos input y output son esencialmente los comandos equivalentes del cap. 8. La principal diferencia es que los canales son declarados globales a los procesos (como con AMP). Sin embargo, cada canal debe tener exactamente un emisor y un receptor.

Los procesos primitivos se combinan en procesos convencionales usando lo que Occam llama *constructores* (es decir, sentencias estructuradas). Estos incluyen constructores secuenciales, un constructor paralelo similar a la sentencia **co**, y una sentencia de comunicación guardada. Un constructor puede ser precedido por declaraciones de variables y canales; el alcance de estos ítems es el constructor. Como veremos, Occam tiene una sintaxis muy distintiva: cada proceso primitivo, constructor, y declaración ocupa una línea; las declaraciones tienen un ":" al final; y el lenguaje impone una convención de indentación.

Los procedures y funciones proveen la única forma de modularización en un programa Occam. Son esencialmente procesos parametrizados y pueden compartir solo canales y constantes. Occam no soporta recursión o cualquier forma de creación o nombrado dinámico. Esto hace que muchos algoritmos sean difíciles de programar. Sin embargo, asegura que un compilador Occam puede determinar exactamente cuántos procesos contiene un programa y cómo se comunican uno con otro. Esto, y el hecho de que distintos constructores no pueden compartir variables, hace posible para un compilador asignar procesos y datos a procesadores en una máquina de memoria distribuida tal como un transputer.

Constructores secuenciales y paralelos

En la mayoría de los lenguajes, el default es ejecutar sentencias secuencialmente; el programador tiene que decir explícitamente cuándo ejecutar sentencias concurrentemente. Occam toma una aproximación distinta: *no hay default !*. En lugar de esto, Occam contiene dos constructores básicos: SEQ para ejecución secuencial y PAR para ejecución paralela. Por ejemplo, el siguiente programa incrementa *x*, luego *y*:

```
INT x,y :
SEQ
  x := x + 1
  y := y + 1
```

Dado que las dos sentencias acceden variables distintas, pueden ser ejecutadas concurrentemente. Esto se expresa por:

```
INT x,y :
PAR
  x := x + 1
  y := y + 1
```

Lo que cambia es el nombre del constructor. Sin embargo, PAR no es tan poderoso como la sentencia **co** pues los procesos Occam no pueden compartir variables.

Occam contiene una variedad de otros constructores. El constructor IF se usa para alternativas: es similar a la sentencia **if** guardada usada en el texto, pero hay dos diferencias: las guardas son evaluadas en el orden en que están listadas, y al menos una guarda **debe** ser true. El constructor CASE es una variante de IF que puede ser usada para sentencias **case**. El constructor WHILE es uno de los mecanismos de iteración; es como la sentencia **while** de Pascal.

Occam también contiene un mecanismo interesante llamado replicador. Es similar a un cuantificador y se usa de manera similar. Por ejemplo, lo siguiente declara un arreglo *a* de 11 elementos e iterativamente asigna un valor a cada elemento:

```
[10] INT a :
SEQ i = 0 FOR 10
  a[i] := i
```

Como vemos, solo el límite superior del arreglo se declara; el límite inferior es implícitamente 0 para todos los arreglos. Las matrices se declaran como arreglos de arreglos.

Un replicador no agrega nueva funcionalidad al lenguaje, solo es una abreviación para un conjunto de componentes. Occam requiere que el límite superior sea una constante, y por lo tanto un replicador siempre puede ser expandido para dar un programa equivalente pero más largo. En el ejemplo anterior, puede usarse PAR en lugar de SEQ pues, si el replicador es expandido, las asignaciones serán distintas textualmente. Sin embargo, en general, un compilador Occam no será capaz de determinar si los procesos son distintos, por ejemplo, si un arreglo es indexado por una variable de control distinta a la del replicador.

Los procesos especificados usando PAR ejecutan con igual prioridad y sobre cualquier procesador. El constructor PRI PAR puede usarse para especificar que el primer proceso tiene la prioridad más alta, el segundo la siguiente, etc. El constructor PLACED PAR puede usarse para especificar explícitamente dónde es ubicado y por lo tanto ejecutado cada proceso.

Dado que Occam requiere que el límite superior de cada replicador sea una constante en tiempo de compilación, un compilador puede determinar exactamente cuántos procesos contiene un programa. Desafortunadamente, esto significa que la cantidad de paralelismo en un programa no puede depender de los datos de entrada. Por ej, si el programador quiere experimentar con distintos números de workers en un algoritmo de workers replicados, entonces debe escribir varios programas apenas distintos. Como alternativa, se puede escribir un único programa para el caso mayor y usar un constructor IF para que algunos procesos no hagan nada, aunque éstos igual serán creados.

Comunicación y Sincronización

Como dijimos, las sentencias en los distintos constructores de Occam no pueden compartir variables. Para comunicarse y sincronizar, deben usar canales. Una declaración de canal tiene la forma:

```
CHAN OF protocol name :
```

El protocolo define el tipo de valores que son transmitidos por el canal. Pueden ser tipos básicos, arreglos de longitud fija o variable, o registros fijos o variantes.

Los canales son accedidos por los procesos primitivos input (?) y output (!). A lo sumo un proceso compuesto puede emitir por un canal, y a lo sumo uno puede recibir por un canal. Dado que el nombrado es estático, un compilador Occam puede forzar este requerimiento. El siguiente programa muestra por pantalla caracteres recibidos desde el teclado:

```
WHILE TRUE
  BYTE ch :
  SEQ
    keyboard ? ch
    ch ! screen
```

Aquí, *keyboard* y *screen* son canales que se asume que están conectados a dispositivos periféricos. Aunque Occam no define mecanismos de E/S como parte del lenguaje, provee mecanismos para ligar canales de E/S a dispositivos. El sistema de desarrollo en transputer, por ejemplo, contiene una librería de procedures de E/S.

El programa anterior usa un buffer simple, *ch*. Puede convertirse en un programa concurrente que use doble buffering empleando dos procesos, uno para leer desde el teclado y uno para escribir en la pantalla. El proceso se comunica usando un canal adicional *comm*; cada uno tiene un carácter local *ch*:

```
CHAN OF BYTE comm :
PAR
  WHILE TRUE
    BYTE ch :
    SEQ
      keyboard ? ch
      comm ! ch
  WHILE TRUE
    BYTE ch :
    SEQ
      comm ? ch
      keyboard ! ch
```

Este programa indica claramente la sintaxis Occam. El requerimiento de que cada ítem esté en una línea separada lleva a programas largos, pero la indentación requerida evita la necesidad de palabras claves para cerrar estructuras.

El constructor ALT soporta comunicación guardada. Cada guarda consta de un input process, una expresión booleana y un input process, o una expresión booleana y un SKIP (lo cual se reduce esencialmente a una expresión booleana pura). Puede usarse un replicador como abreviatura, por ejemplo, para esperar recibir entrada de uno de un arreglo de canales. Por ejemplo, lo siguiente implementa un alocador de recursos simple:

```
ALT i = 0 FOR n
  avail > 0 & acquire[i] ? unitid
    SEQ
      avail := avail - 1          -- and select unit to allocate
      reply[i] ! unitid
  release[i] ? unitid
```

```
        avail := avail + 1          -- and return unit
```

Aquí, *acquire*, *reply* y *release* son arreglos de canales, con un elemento para cada cliente *i*. Se necesitan arreglos pues un canal puede ser usado por solo dos procesos. Occam también no permite mensajes nulos (señales) de modo que se debe enviar algún valor por el canal *acquire* aunque no se use este valor.

No se permiten output processes en guardas de un constructor ALT. Como dijimos en el cap. 8, esto hace que algunos algoritmos sean duros de programar, pero simplifica enormemente la implementación. Esto último es especialmente importante pues Occam es el lenguaje de máquina del transputer.

Las guardas en un ALT son evaluadas en orden indefinido. El constructor PRI ALT puede usarse para forzar que las guardas sean evaluadas en el orden deseado.

Occam también contiene una facilidad de timer. Un timer es una clase especial de canal. Un proceso hardware está siempre listo para enviar hacia algún canal timer declarado en un programa. Si un proceso declara el canal timer *myclock*, puede leer la hora ejecutando:

myclock ? *time*

El proceso también puede demorarse hasta que el reloj alcance un cierto valor ejecutando:

myclock ? *time* AFTER *value*

Aquí, *value* es un tiempo absoluto, no un intervalo. Un timer también puede usarse en una guarda en un constructor ALT; esto puede usarse para programar un interval timer.

Ejemplo: Números Primos

Damos un programa Occam completo para generar los primeros *n* primos usando la criba de Eratóstenes. Como en el cap. 8, la solución usa un pipeline de procesos filtro. Ilustra el uso de procedures Occam (PROCs) para describir templates para los procesos; los argumentos a cada proceso especifican los canales que el proceso usa.

```
VAL INT n IS 50 :                -- número de primos a generar
VAL INT limit IS 1000 :          -- rango de números a chequear
[n-2] CHAN OF INT link :         -- links entre procesos filtro
[n-1] CHAN OF INT prime :        -- canales al proceso Print
CHAN OF INT display :            -- display de salida conectado al dispositivo 1
PLACE display AT 1 :
PROC Starter (CHAN OF INT out, print)       -- genera números impares
  INT i :
  SEQ
    print ! 2
    i := 3
    WHILE i < limit
      SEQ
        out ! i
        i := i + 2 :
PROC Sieve (CHAN OF INT in, out, print)     -- filtra un primo
  INT p, next :
  SEQ
    in ? p                       -- p es un primo
    print ! p
    WHILE TRUE
      SEQ
        in ? next
        IF
```

```
            (next/p) <> 0          -- es evaluado primero
              out ! next
            TRUE                   -- IF necesita que alguna guarda sea true
              SKIP :
PROC Ender (CHAN OF INT in, print)       -- consume el resto de los números
  INT p :
  SEQ
    in ? p
    print ! p
    WHILE TRUE
      in ? p :
PROC Printer ([ ] CHAN OF INT value)    -- imprime cada primo, en orden
  INT p :
  SEQ i = 0 FOR SIZE value
    SEQ
      value[i] ? p
      display ! p :
PAR                            -- comienza la ejecución de todos los procesos
  Starter(link[0], prime[0])
  PAR i = 1 FOR n-2
    Sieve(link[i-1], link[i], prime[i])
  Ender(link[n-1], prime[n-1])
  Printer(prime)
```

La constante *n* especifica cuántos primos generar; *limit* es un valor al menos tan grande como el n-ésimo primo. Los procedures declaran templates para los procesos filtro y el proceso printer. El pipeline consta de una instancia de *Starter*, *n*-2 instancias de *Sieve*, y una instancia de *Ender*. Cada proceso produce un primo. Los *n*-1 canales *link* se usan para conectar el pipeline de procesos. Los *n* canales *print* se usan para enviar primos al proceso *Printer*. El canal *display* se usa para imprimir los primos en el display de salida standard, el cual por convención es el dispositivo físico 1.

El código de inicialización al final del programa inicia los procesos. El proceso *Printer* se pasa como un arreglo de canales, uno por cada proceso filtro. Se necesita un arreglo pues solo dos procesos pueden usar un canal. El código de inicialización usa constructores PAR anidados; esto es para usar un replicador en el constructor PAR que activa los procesos *Sieve*.

Como están programados en el ejemplo, los procesos *Sieve* y *Ender* ejecutan loops permanentes. Por lo tanto, cuando el programa como un todo termina, estos procesos quedan bloqueados esperando una entrada adicional. Podemos cambiar el programa para que cada proceso termine normalmente por ejemplo enviando un valor centinela por cada canal *in*.

ADA: RENDEZVOUS

A diferencia de los otros lenguajes descriptos en este capítulo, Ada es el resultado de una competencia de diseño internacional. En consecuencia, es el lenguaje de programación concurrente más conocido. Las características de concurrencia de Ada son una parte importante del lenguaje (y son críticas para su uso en programación de embedded systems que controlan tráfico aéreo, submarinos, etc. Sin embargo, Ada también contiene un conjunto de mecanismos increíblemente grande y rico para programación secuencial.

La versión inicial de Ada se diseñó en 1977-78 y refinada en 1978-79. La versión corriente de Ada fue standarizada en 1983. Con respecto a la programación concurrente, los principales mecanismos del lenguaje son las tasks (procesos) y el rendezvous. Ada también contiene mecanismos de timing para escribir STR. Daremos una idea de los mecanismos principales de concurrencia del lenguaje, así como algunos de los mecanismos secuenciales y de estructura de programa.

Componentes de programa

Un programa Ada se construye a partir de subprogramas, packages y tasks. Un subprograma es un procedure o función, un package es una colección de declaraciones, y una task es esencialmente un proceso. Cada uno tiene una parte de especificación y un cuerpo. La parte de especificación declara objetos visibles; el cuerpo contiene declaraciones y sentencias locales adicionales. (La especificación y el cuerpo pueden compilarse separadamente). Los subprogramas y packages también pueden ser genéricos, es decir, pueden ser parametrizados por tipos de datos.

La forma más común de una especificación de task es:

```
task identifier is
  declaraciones de entry
end;
```

Las declaraciones de entry son similares a las declaraciones **op**; definen operaciones servidas por la task. Una task sin entry points tiene una especificación abreviada. La especificación de una task que controla dispositivos de hardware puede incluir lo que se llaman cláusulas de representación, por ejemplo, para mapear un entry a una interrupción de hardware.

La forma más común de un cuerpo de task es:

```
task body identifier is
  declaraciones locales
begin
  sentencias
end identifier;
```

El body también puede contener manejadores de excepción.

Una task debe ser declarada dentro de un subprograma o package. El programa concurrente más simple en Ada es un único procedure que contiene especificaciones y cuerpos de task:

```
procedure identifier is
  tasks y otras declaraciones
begin
  sentencias secuenciales
end identifier;
```

Las declaraciones son *elaboradas* una a la vez, en el orden en el cual aparecen. Elaborar una declaración de task crea una instancia de la task. Después de que todas las declaraciones son elaboradas, las sentencias secuenciales en el subprograma comienzan la ejecución como una task anónima. Aunque Ada desaconseja el uso de variables compartidas, éstas pueden usarse. Si las secuencias secuenciales inicializan estas variables, el programador tiene que asegurar que otras tasks se demoran hasta que las variables compartidas fueron inicializadas.

Una especificación de task define una única tarea. Una instancia del correspondiente task body se crea en el bloque en el cual se declara la task. Ada también soporta arreglos de tareas, pero de manera distinta a la de la mayoría de otros lenguajes. En particular, el programador primero declara un task type y luego declara un arreglo de instancias de ese tipo. El programador también puede usar task types en conjunción con punteros (que Ada llama access types) para crear tasks dinámicamente. Sin embargo, Ada no provee ningún mecanismo para asignar tasks a procesadores y por lo tanto para controlar el layout de un programa distribuido; si esto es necesario, tiene que ser expresado en un lenguaje de configuración específico de la implementación.

Comunicación y Sincronización

El rendezvous es el único mecanismo de sincronización en Ada; cualquier otra cosa, tal como una bolsa de tasks o comunicación buffereada, tiene que ser programada usándolo. Rendezvous también es el mecanismo de comunicación primario (las tasks también pueden compartir variables). Como dijimos, las declaraciones de entry son muy similares a las declaraciones **op**. Tienen la forma:

entry identifier(formales)

Los parámetros del entry en Ada son pasados por copy-in (**in**, el default), copy-out (**out**), o copy-in/copy-out (**in out**). Ada también soporta arreglos de entries, llamados familias de entry.

Si la task T declara el entry E, otras tasks en el alcance de la especificación de T pueden invocar a E por una sentencia **call**:

call T.E(reales)

Como es usual, la ejecución de **call** demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción).

La task que declara un entry sirve llamados de ese entry por medio de la sentencia **accept**. Esto tiene la forma general:

```
accept E(formales) do
  lista de sentencias
end;
```

La ejecución de **accept** demora la tarea hasta que haya una invocación de E, copia los argumentos de entrada en los formales de entrada, luego ejecuta la lista de sentencias. Cuando la lista de sentencias termina, los formales de salida son copiados a los argumentos de salida. En ese punto, tanto el llamador como el proceso ejecutante continúan. Una sentencia **accept** es como una input statement con una guarda, sin expresión de sincronización y sin expresión de scheduling.

Para controlar el no determinismo, Ada provee tres clases de sentencias **select**: wait selectivo, entry call condicional, y entry call timed. La sentencia wait selectiva soporta comunicación guardada. La forma más común de esta sentencia es:

select when $B_1 \Rightarrow$ sentencia accept; sentencias
or ...
or **when** $B_n \Rightarrow$ sentencia accept; sentencias
end select

Cada línea (salvo la última) se llama *alternativa*. Las B_i son expresiones booleanas, y las cláusulas when son opcionales. Una alternativa se dice que está *abierta* si B_i es true o se omite la cláusula **when**.

Esta forma de wait selectivo demora al proceso ejecutante hasta que la sentencia **accept** en alguna alternativa abierta pueda ser ejecutada, es decir, haya una invocación pendiente del entry nombrado en la sentencia **accept**. Dado que cada guarda B_i precede una sentencia **accept**, no puede referenciar los parámetros de un entry call. Además, Ada no provee expresiones de scheduling, lo cual hace difícil resolver algunos problemas de sincronización y scheduling.

La sentencia wait selectiva puede contener una alternativa opcional **else**, la cual es seleccionada si ninguna otra alternativa puede serlo. En lugar de la sentencia **accept**, el programador puede también usar una sentencia **delay** o una alternativa **terminate**. Una alternativa abierta con una sentencia **delay** es seleccionada si transcurrió el intervalo de delay; esto provee un mecanismo de timeout. La alternativa **terminate** es seleccionada esencialmente si todas las tasks que rendezvous con esta terminaron o están esperando una alternativa **terminate**.

Estas distintas formas de sentencias wait selectiva proveen una gran flexibilidad, pero también resultan en un número algo confuso de distintas combinaciones. Para “empeorar” las cosas, hay dos clases adicionales de sentencia **select**.

Un entry call condicional se usa si una task quiere hacer poll de otra. Tiene la forma:

```
select entry call; sentencias
else   sentencias
end select
```

El entry call es seleccionado si puede ser ejecutado inmediatamente; en otro caso, se selecciona la alternativa **else**.

Un entry call timed se usa si una task llamadora quiere esperar a lo sumo un cierto intervalo de tiempo. Su forma es similar a la de un entry call condicional:

```
select entry call; sentencias
or     sentencia de delay; sentencias
end select
```

En este caso, el entry call se selecciona si puede ser ejecutado antes de que expire el intervalo de delay. Esta sentencia soporta timeout, en este caso, de un **call** en lugar de un **accept**.

Ada provee unos pocos mecanismos adicionales para programación concurrente. Las tasks pueden compartir variables; sin embargo, no pueden asumir que estas variables son actualizadas excepto en puntos de sincronización (por ej, sentencias de rendezvous). La sentencia **abort** permite que una tarea haga terminar a otra. Hay un mecanismo para setear la prioridad de una task. Finalmente, hay atributos que habilitan para determinar cuándo una task es llamable o ha terminado o para determinar el número de invocaciones pendientes de un entry.

Ejemplo: Filósofos

Esta sección presenta un programa Ada para el problema de los filósofos. El programa ilustra el uso de tasks, rendezvous, compilación separada, E/S, declaraciones, y sentencias secuenciales.

Programa principal:

```
with TEXT_IO; use TEXT_IO;
procedure DINING_PHILOSOPHERS is
  subtype ID is INTEGER range 1..5;
  task WAITER is
    entry NEED(I : in ID);
    entry PICKUP(ID)( );      -- un entry por filósofo
    entry PUTDOWN(I : in ID);
  end
  task body WAITER is separate;
  task type PHILOSOPHER is
    entry INIT(WHO : in ID);
  end;
  DP : array(ID) of PHILOSOPHER;    -- las 5 tasks filósofos
  ROUNDS : INTEGER;    -- número de rondas para pensar y comer
  task body PHILOSOPHER
    MYID : ID;
  begin
    accept INIT(WHO);      -- toma la identidad
      MYID := WHO;
    end
    for J in 1..ROUNDS loop
      -- “piensa”
```

```
            call WAITER.NEED(MYID);       -- le dice al waiter I que quiere comer
            call WAITER.PICKUP(MYID)( );   -- espera tener permiso
            -- "come"
            call WAITER.PUTDOWN(MYID);
          end loop;
        end PHILOSOPHER;
      begin
        GET(ROUNDS)      -- lee el número de rondas desde la entrada standard
        for J in 1..ID loop      -- le da a cada filósofo su índice
          call DP(J).INIT(J);
        end loop;
      end DINING_PHILOSOPHERS;
```

Tarea waiter:

```
      separate (DINING_PHILOSOPHERS)
      task body WAITER is
        WANT : array (ID) of BOOLEAN;    -- cuales waiters quieren comer
        EATING : array (ID) of BOOLEAN;    -- cuales waiters están comiendo
        WHO : INTEGER
      begin
        for J in ID loop                        -- inicializa los arreglos
          WANT(J) :+ FALSE; EATING(J) := FALSE;
        end loop;
        loop
          select
            accept NEED(I : in ID) do   -- el filósofo I necesita los tenedores
              WHO := I;
            end;
            -- mira si el filósofo I (WHO) puede comer
            if not (EATING(WHOΘ1) or EATING(WHO⊕1)) then
              accept PICKUP(WHO)( );
              EATING(WHO) := TRUE;
            else
              WANT(WHO) := TRUE;
            end if
          or
            accept PUTDOWN(I : in ID) do     -- el filósofo I terminó de comer
              EATING(I) := FALSE; WHO := I;
            end
            -- chequea los vecinos para ver si quieren y pueden comer
            if WANT(WHOΘ1) and not EATING(WHOΘ2) then
              accept PICKUP(WHOΘ1)( );
              EATING(WHOΘ1) := TRUE; WANT(WHOΘ1) := FALSE;
            end if;
            if WANT(WHO⊕1) and not EATING(WHO⊕2) then
              accept PICKUP(WHO⊕1)( );
              EATING(WHO⊕1) := TRUE; WANT(WHO⊕1) := FALSE;
            end if;
          or
            terminate;    -- termina cuando todos los filósofos terminaron
          end select;
        end loop;
      end WAITER;
```

La primera parte contiene el procedure principal DINING_PHILOSOPHERS. Antes del procedure hay una cláusula **with** y una cláusula **use**. La cláusula **with** dice que este procedure depende de los objetos en TEXT_IO, el cual es un package predefinido. La cláusula **use** hace directamente visibles los nombres de los objetos exportados por ese package (es decir que no tienen que ser calificados con el nombre del package).

El procedure principal primero declara ID como un entero que varía entre 1 y 5, los índices de los 5 filósofos. La especificación de la task WAITER declara tres entries: NEED, PICKUP, y PUTDOWN. PICKUP es una familia (arreglo) de entries. Cada filósofo llama a NEED y PUTDOWN; cada uno usa un elemento diferente de PICKUP para sincronizar con el waiter. En este ejemplo, el cuerpo de WAITER es compilado separadamente.

La especificación de PHILOSOPHER es un task type de modo que podemos declarar un arreglo DP de 5 de tales tasks. El body de cada PHILOSOPHER es un loop que ejecuta ROUNDS iteraciones. ROUNDS es una variable compartida; el código de inicialización al final del procedure principal llama al procedure GET para asignar un valor a ROUNDS. Cada filósofo tiene un entry, INIT, que el código de inicialización llama para pasarle al filósofo su índice en el arreglo de tasks. No hay forma en Ada para que un elemento de un arreglo de tasks determine su propio índice. Sin embargo, este uso de rendezvous asegura que la variable compartida ROUNDS no es leído hasta que fue inicializada.

La segunda parte del programa contiene el cuerpo de la task WAITER. La primera línea indica que la especificación de WAITER está contenida en otra unidad de compilación. La task WAITER es mucho más complicada que el proceso *Waiter* del cap. 9. Esto es porque las condiciones **when** en la sentencia **select** no pueden referenciar parámetros del entry. En consecuencia, un filósofo primero tiene que llamar a NEED para decirle al waiter que quiere comer, luego tiene que llamar a su elemento del arreglo de entries PICKUP para esperar que le den permiso para comer.

Cuando WAITER acepta un llamado de NEED, registra quién llamó y luego termina la sentencia **accept**; esto desbloquea al filósofo llamador que puede así llamar a PICKUP(I). El WAITER luego determina si el filósofo puede comer ahora. Si es así, WAITER acepta el llamado de PICKUP(I); en otro caso, WAITER registra que el filósofo I quiere comer. El entry PICKUP(I) se usa solo para sincronización, de modo que la sentencia **accept** que lo sirve no tiene una lista de sentencias asociada.

Después de comer, un filósofo llama a PUTDOWN. Cuando WAITER acepta este llamado, chequea si algún vecino quiere comer y si puede hacerlo. En este caso, la sentencia **accept** que sirve PUTDOWN podría comprender la alternativa select entera (es decir, el end del **accept** podría estar después de las dos sentencias **if**. Sin embargo, está antes pues no hay necesidad de demorar al filósofo que llamó a PUTDOWN.

Para este problema, podríamos haber servido los entries PICKUP en 5 alternativas adicionales de la sentencia **select**, con cada una guardada por la condición apropiada. Por ejemplo, una de estas alternativas podría ser:

```
when not (EATING(1) or EATING(3)) ⇒
  accept PICKUP(2)( ); EATING(2) := TRUE;
```

Necesitaríamos usar 5 alternativas distintas pues cada guarda es distinta y Ada no provee algo como cuantificadores para simplificar la expresión de condiciones muy relacionadas.

SR: PRIMITIVAS MULTIPLES

La primera versión de SR (Sinchronizing Resources) fue diseñada al final de los 70s. El lenguaje evolucionó desde un lenguaje puramente distribuido con múltiples primitivas de pasaje de mensajes a uno que también contiene mecanismos para programación de computaciones de memora compartida. En consecuencia, SR puede ser usado para implementar directamente casi todos los algoritmos de este libro. De hecho, las notaciones secuenciales y concurrentes usadas son idénticas a las de SR. Aunque SR contiene una gran variedad de mecanismos, están basados en un número pequeño de conceptos ortogonales. También, los mecanismos secuenciales y concurrentes son integrados cuidadosamente de modo que cosas similares se hacen de maneras similares.

Ya discutimos e ilustramos muchos aspectos de SR sin decirlo realmente. Esta sección se concentra en aspectos adicionales: estructura de programa, nombrado dinámico, y creación y ubicación dinámica.

Componentes de programa

Un programa SR es construido a partir de lo que llamamos recursos y globales. Un recurso es un patrón para un módulo; su estructura es similar a la de un **module**:

```
resource name
  import specification
  operation and type declarations
body name(formales)
  declaraciones locales
  código de inicialización
  procedures y procesos
  código de finalización
end name
```

Cada recurso es compilado separadamente; la especificación y cuerpo de un recurso también pueden ser compilados separadamente. Un recurso contiene una o más import specifications si hace uso de las declaraciones exportadas desde otros recursos o desde globales. Las declaraciones y el código de inicialización en el cuerpo pueden estar entremezclados; esto soporta arreglos dinámicos y permite al programador controlar el orden en el cual las variables son inicializadas y los procesos son creados.

Las instancias de recursos son creadas dinámicamente por medio de la sentencia **create**. Por ejemplo, ejecutando

```
rcap := create name(reales)
```

pasa los reales (por valor) a una nueva instancia del recurso *name* y luego ejecuta el código de inicialización de ese recurso. Cuando el código de inicialización termina, una *resource capability* es retornada por **create** y asignada a la variable *rcap*. Esta variable puede luego ser usada para invocar operaciones exportadas por el recurso o para destruir la instancia.

Una componente global es esencialmente una instancia simple no parametrizada de un recurso. Se usa para declarar cosas como tipos, variables, operaciones y procedures que son compartidos por recursos. Una instancia de un global es almacenada en cada máquina virtual (espacio de direcciones) que la necesita. En particular, cuando se crea un recurso, todos los globales que importa son creados implícitamente si todavía no fueron creados.

Por default, la ejecución de **create** pone un nuevo recurso en la misma máquina virtual que la del recurso que lo creó. Para construir una implementación distribuida de un programa, se puede agregar "**on** *machine*" a la sentencia **create**; *machine* es el nombre simbólico de otra máquina o una capability para una máquina virtual creada dinámicamente. Así SR, a diferencia de Ada, le da al programador control completo sobre cómo los recursos son mapeados en máquinas y este mapeo puede depender de la entrada al programa.

Un programa SR siempre contiene un recurso principal distinguido. La ejecución de un programa comienza con la creación implícita de una instancia de este recurso. Luego el código de inicialización en el recurso principal es ejecutado; típicamente crea instancias de otros recursos.

Todos los recursos son creados dinámicamente. También pueden ser destruidos dinámicamente por medio de la sentencia **destroy**. Si *rcap* es una capability para una instancia de un recurso, la ejecución de

```
destroy rcap
```

detiene cualquier actividad en esa instancia, ejecuta el código de finalización, y luego libera el almacenamiento alocado para esa instancia.

Un programa SR termina cuando todo proceso terminó o está bloqueado (también hay una sentencia **stop** explícita). En ese punto, el sistema de run-time ejecuta el código de finalización del recurso principal y luego el código de finalización en los globales. Esto provee una forma de que el programador regane el control para imprimir resultados e información de timing.

Comunicación y Sincronización

El atributo más distintivo de SR es su variedad de mecanismos de comunicación y sincronización. Los procesos en el mismo recurso pueden compartir variables, como pueden los recursos en el mismo espacio de direcciones (a través del uso de globales). Los procesos pueden comunicarse y sincronizar también usando todas las primitivas descriptas en el cap. 9: semáforos, AMP, RPC, y rendezvous. Así, SR puede ser usado para implementar programas concurrentes tanto para multiprocesadores de memoria compartida como para sistemas distribuidos.

Las operaciones son declaradas en declaraciones **op**, que tienen la forma dada en el cap. 9. Tales declaraciones pueden aparecer en especificaciones de recurso, en cuerpos de recurso, y aún dentro de procesos. Una operación declarada dentro de un proceso es llamada *operación local*. El proceso declarante puede pasar una capability para una operación local a otro proceso, el cual puede entonces invocar la operación. Esto soporta continuidad conversacional.

Una operación es invocada usando **call** sincrónico o **send** asincrónico. Para especificar qué operación invocar, una sentencia de invocación usa una capability de operación o un campo de una capability de recurso. Dentro del recurso que la declara, el nombre de una operación es de hecho una capability, de modo que una sentencia de invocación puede usarla directamente. Las capabilities de recurso y operación pueden ser pasadas entre recursos, de modo que los paths de comunicación pueden variar dinámicamente.

Una operación es servida o por un procedure (**proc**) o por sentencias de entrada (**in**). Un nuevo proceso es creado para servir cada llamado remoto de un **proc**; los llamados desde dentro del mismo espacio de direcciones son optimizados de modo que el llamador por sí mismo ejecuta el cuerpo del procedure. Todos los procesos en un recurso ejecutan concurrentemente, al menos conceptualmente.

La sentencia de entrada soporta rendezvous. Tiene la forma mostrada en el cap. 9 y puede tener tanto expresiones de sincronización como de scheduling que dependen de parámetros. La sentencia de entrada también puede contener una cláusula opcional **else**, que es seleccionada si ninguna otra guarda tiene éxito.

Varios mecanismos en SR son abreviaciones para usos comunes de operaciones. Una declaración **process** es una abreviación para una declaración **op** y un **proc** para servir invocaciones de la operación. Una instancia del proceso se crea por un **send** implícito cuando el recurso es creado. El cuerpo de un **process** con frecuencia es un loop permanente. (También pueden declararse arreglos de procesos). Una declaración **procedure** es una abreviación para una declaración **op** y un **proc** para servir invocaciones de la operación.

Dos abreviaciones adicionales son la sentencia **receive** y los semáforos. En particular, **receive** abrevia una sentencia de entrada que sirve una operación y que solo almacena los argumentos en variables locales. Una declaración de semáforo (**sem**) abrevia la declaración de una operación sin parámetros. La sentencia **P** es un caso especial de **receive**, y la sentencia **V** un caso especial de **send**.

SR también provee algunas sentencias adicionales útiles. La sentencia **reply** es usada por un **proc** para retornar valores, tal como links de comunicación, a su llamador y luego continuar la ejecución. La sentencia **forward** puede ser usada para pasar una invocación a otro proceso para su servicio; en este caso el llamador se mantiene bloqueado. Finalmente, SR contiene una forma restringida de **co** que puede ser usada para crear recursos o invocar operaciones en paralelo.

LINDA: ESTRUCTURAS DE DATOS DISTRIBUIDAS

Linda provee un enfoque de la programación concurrente que es bastante distinto del de los otros considerados. Primero, Linda no es en si mismo un lenguaje. Es un pequeño número de primitivas que son usadas para acceder lo que llamamos espacio de tupla (tuple space, TS). Cualquier lenguaje de programación secuencial puede ser aumentado con las primitivas de Linda para obtener una variante de programación concurrente de ese lenguaje.

Segundo, Linda generaliza y sintetiza aspectos de variables compartidas y AMP. El TS es una memoria compartida, asociativa, consistente de una colección de registros de datos tagged llamados tuplas. TS es como un canal de comunicación compartido simple, excepto que las tuplas no están ordenadas. La operación para depositar una tupla (**out**) es como un **send**, la operación para extraer una tupla (**in**) es como un **receive**, y la operación para examinar una tupla (**rd**) es como una asignación desde variables compartidas a locales. Una cuarta operación, **eval**, provee creación de procesos. Las dos operaciones finales, **inp** y **rdp**, proveen entrada y lectura no bloqueante.

Aunque TS es compartido por procesos, puede ser (y fue) implementado por partes distribuidas entre procesadores en un multicomputador o una red. Así, TS puede usarse para almacenar estructuras de datos distribuidas, y distintos procesos pueden operar concurrentemente sobre distintos elementos de la estructura de datos. Esto soporta directamente el paradigma de worker replicado para interacción entre procesos. Linda también soporta programación paralela de datos y MP, aunque no tan elegantemente.

Linda fue concebido al inicio de los 80s. La propuesta inicial tenía tres primitivas; las otras se agregaron a mediados y fines de los 80s. Varios lenguajes fueron aumentados con las primitivas de Linda, incluyendo C y Fortran. En los ejemplos, se usa C-Linda.

Tuple Space e Interacción de procesos

El TS consta de una colección no ordenada de tuplas de datos pasivos y tuplas de procesos activos. Las tuplas de datos son registros tagged que contienen el estado compartido de una computación. Las tuplas proceso son rutinas que ejecutan asincrónicamente. Interactúan leyendo, escribiendo y generando tuplas de datos. Cuando una tupla proceso termina, se convierte en una tupla de datos.

Cada tupla de datos en TS tiene la forma:

("tag", $value_1$, ..., $value_n$)

El tag es un string literal que se usa para distinguir entre tuplas que representan distintas estructuras de datos. Los $value_i$ son cero o más valores de datos, por ejemplo, enteros, reales o arreglos.

Linda define tres primitivas básicas para manipular tuplas de datos: **out**, **in**, y **rd**. Las tres son atómicas (aparecen como operaciones indivisibles). Un proceso deposita una tupla en TS ejecutando:

out("tag", $expr_1$, ..., $expr_n$)

La ejecución de **out** termina una vez que las expresiones fueron evaluadas y la tupla de datos resultante fue depositada en TS. La operación **out** es similar a una sentencia **send**, excepto que la tupla se almacena en un TS no ordenado en lugar de ser agregada a un canal específico.

Un proceso extrae una tupla de datos de TS ejecutando:

in("tag", $field_1$, ..., $field_n$)

Cada $field_i$ o es una expresión o un parámetro formal de la forma *? var* donde *var* es una variable local en el proceso ejecutante. Los argumentos para **in** son llamados *template*. El proceso que ejecuta **in** se demora hasta que TS contenga al menos una tupla que matchee el template, y luego remueve una de TS.

Un template *t* matchea una tupla de datos *d* si: (1) los tags son idénticos, y *t* y *d* tienen el mismo número de campos, (2) las expresiones en *t* son iguales al valor correspondiente en *d*, y (3) las variables en *t* tienen el mismo tipo que el valor correspondiente en *d*. Después de que una tupla matching es removida de TS, los parámetros formales en el template son asignados con los valores correspondientes de la tupla. Así, **in** es como una sentencia **receive**, con el tag y los valores en el template sirviendo para identificar el canal.

Dos ejemplos simples pueden ayudar a simplificar estos conceptos. Primero, podemos simular un semáforo en Linda como sigue. Sea "sem" el nombre simbólico de un semáforo. Entonces la operación **V** es simplemente **out**("sem"), y la operación **P** es simplemente **in**("sem"). El valor del semáforo es el número de tuplas "sem" en TS. Para simular un arreglo de semáforos, usaríamos un campo adicional para representar el índice del arreglo, por ejemplo:

```
out("forks", i)        # V(forks[i])
in("forks", i)         # P(forks[i])
```

El template en la sentencia **in** matchea cualquier tupla que tenga el mismo tag "forks" y el mismo valor *i*.

La tercera primitiva básica de tuplas de datos es **rd**, que se usa para examinar tuplas. Si *t* es un template, la ejecución de **rd**(t) demora al proceso hasta que TS contiene una tupla de datos matching. Como con **in**, las variables en *t* luego son asignadas con los valores correspondientes de los campos de la tupla de datos. Sin embargo, la tupla permanece en TS.

También hay variantes no bloqueantes de **in** y **rd**. Las operaciones **inp** y **rdp** son predicados que retornan true si hay una tupla matching en TS, y falso en otro caso. Si retornan true, tienen el mismo efecto que **in** y **rd** respectivamente. Estas variantes proveen una manera de que un proceso haga poll sobre TS.

Usando las tres primitivas básicas, podemos modificar y examinar una variable "global". Por ejemplo, podemos implementar un simple contador barrier para *n* procesos como sigue. Primero, algún proceso inicializa la barrera depositando un contador en TS:

```
out("barrier", 0)
```

Cuando un proceso alcanza un punto de sincronización barrier, primero incrementa el contador ejecutando:

```
in("barrier", ? counter)    # extrae la tupla y toma el valor del contador
out("barrier", counter+1) # incrementa el contador y lo vuelve a poner
```

Luego el proceso espera a que los *n* procesos arriben ejecutando:

```
rd("barrier", n)
```

Esta sentencia demora al proceso hasta que la parte del contador de la tupla barrera sea exactamente *n*.

La sexta y última primitiva Linda es **eval**, que crea una tupla proceso. Esta operación tiene la misma forma que **out**:

eval("tag", $expr_1$, ..., $expr_n$)

La diferencia es que un proceso es forked para evaluar las expresiones. Una de las expresiones es típicamente un llamado a procedure o función. Cuando el proceso forked termina, la tupla se vuelve una tupla de datos pasiva.

La primitiva **eval** provee el medio por el cual la concurrencia es introducida en un programa Linda. Como ejemplo, consideremos la siguiente sentencia concurrente:

co i := 1 **to** n → a[i] := f(i) **oc**

Esta sentencia evalúa *n* llamados de f en paralelo y asigna los resultados al arreglo compartido *a*. El código C-Linda correspondiente es:

```
for ( i = 1; i ≤ n; i++ )
  eval("a", i, f(i));
```

Esto forks *n* tuplas proceso; cada una evalúa un llamado de f(i). Cuando una tupla proceso termina, se convierte en una tupla de datos que contiene el nombre del arreglo, un índice y el valor de ese elemento de *a*.

TS también puede usarse para implementar pasaje de mensajes punto-a-punto convencional. Por ejemplo, un proceso puede enviar un mensaje a un canal *ch* ejecutando:

out("ch", expresiones)

Nuevamente, el nombre de un objeto compartido se convierte en un tag en una tupla. Otro proceso puede recibir el mensaje ejecutando:

in("ch", variables)

Sin embargo, esta sentencia extrae cualquier tupla matching. Si es importante que los mensajes sean recibidos en el orden en que se enviaron, podemos usar contadores como sigue. Asumimos que *ch* tiene un emisor y un receptor. Luego el emisor puede agregar un campo contador adicional a cada tupla "ch"; el valor en este campo es 1 para el primer mensaje, 2 para el segundo, etc. El receptor luego usa su propio contador para matchear el primer mensaje, luego el segundo, etc. Si el canal tiene múltiples emisores o receptores, los contadores pueden por sí mismos ser almacenados en TS; serían leídos e incrementados como el contador barrier dado antes.

Para resumir, TS contiene tuplas de datos que son producidas por **out** y **eval** y examinadas por **in**, **rd**, **inp** y **rdp**. Dado que TS es compartido, todos los procesos pueden examinarlo. Aunque cada operación sobre TS debe aparecer como indivisible, distintos procesos pueden examinar distintas tuplas concurrentemente. Esto soporta tanto los estilos de programación de datos paralelos como de workers replicados. Sin embargo, TS es no protegido (cualquier proceso puede extraer o examinar cualquier tupla. Más aún, los tags son strings lineales de caracteres; no hay jerarquía en el espacio de nombre y ningún sentido de alcance. Estos atributos de TS son malos para abstracción y posterior desarrollo de grandes programas.

Otra desventaja de TS es que la performance de las operaciones es en general mucho peor que la de cualquier mecanismo de variables compartidas o MP descriptos. En una máquina de memoria compartida, TS puede ser almacenado en memoria compartida por todos los procesadores y por lo tanto todos los procesos. Sin embargo, las tuplas no pueden ser actualizadas en su lugar; tienen que ser extraídas, modificadas y luego depositadas nuevamente en TS. Más aún, implementar las operaciones **rd** e **in** requiere hacer un lookup asociativo para una tupla matching. La situación es peor en una máquina de memoria distribuida pues TS estaría desparramado en las máquinas. Por ejemplo, una operación **out** o **eval** podría poner una tupla en la memoria local; pero luego una operación **rd** o **in** en general tendría que examinar todas las memorias. Sin embargo, con soporte de hardware para broadcast, es posible tener una performance razonable.

COMPARACION Y PERFORMANCE

Los capítulos previos examinaron varios mecanismos para comunicación y sincronización entre procesos. Este capítulo dio una visión de 5 lenguajes que ilustran distintas combinaciones de estos mecanismos. Los 5 lenguajes también tienen otras diferencias:

* Occam es un lenguaje muy Espartano con una sintaxis peculiar; Ada es extremadamente difuso (verbose).
* Turing Plus soporta computaciones de memoria compartida; Occam, Ada y Linda soportan computaciones distribuidas; SR soporta ambos.
* Occam tiene un número estático de procesos; Turing Plus, SR, y Linda soportan creación dinámica de procesos; Ada está en el medio (las tasks son estáticas, pero pueden usarse punteros para crear instancias de task types)
* Linda tiene un número muy chico de primitivas; SR tiene muchas (pero todas son variaciones en las maneras de invocar y servir operaciones)
* Turing Plus, Ada, y SR separan la especificación de una componente de su implementación; Occam y Linda no.
* SR da al programador control sobre dónde se ubican las componentes de una computación distribuida; los otros lenguajes no.

Lo que quizá no es tan obvio es que los distintos mecanismos de sincronización y comunicación también tienen grandes diferencias en performance. Naturalmente, se tarda más en comunicarse usando MP (especialmente a través de una red) que haciendolo con variables compartidas. Pero hay diferencias de performance mayores entre los distintos mecanismos de MP. Esto afecta la performance de las aplicaciones.

Esta sección describe experimentos para medir la performance de varios mecanismos de interacción entre procesos: procedure call, semáforos, creación de procesos, AMP, SMP, RPC, y rendezvous. Primero se describen cómo se realizaron los experimentos y luego se presentan e interpretan los resultados.

Experimentos de Performance

Usamos el lenguaje SR para los experimentos pues incluye todos los mecanismos. Otros mostraron independientemente que la performance de cada mecanismo en SR es comparable con la de los lenguajes que contienen solo un subconjunto de los mecanismos de SR.

Para proveer una base para interpretar los resultados, primeros resumimos cómo está implementado SR. La implementación SR es construida sobre UNIX. Tiene 3 grandes componentes: compilador, linker y sistema de run-time (RTS). El compilador SR genera código C, el cual es pasado a través de un compilador C para producir código de máquina. El linker SR combina el código máquina de distintos recursos, librerías standard, el RTS, y un manager de ejecución remota (si es necesario) para producir un programa ejecutable.

Durante la ejecución, un programa SR consta de una o más máquinas virtuales (espacios de direcciones). Cada máquina virtual ejecuta dentro de un único proceso UNIX; contiene almacenamiento para variables de programa, código secuencial, y una instancia del RTS. Los RTSs en distintas máquinas virtuales interactúan usando sockets UNIX, los cuales a su vez usan una red de comunicación si las máquinas virtuales están en distintas máquinas físicas.

El RTS es una colección de primitivas que implementan procesos y los mecanismos SR de interacción entre procesos. En particular, el RTS contiene descriptores para procesos y clases de operación y primitivas para invocar y servir operaciones. El RTS también contiene descriptores de recurso y primitivas adicionales para manejo de memoria, manejo de recursos, semáforos, máquinas virtuales, y E/S.

Para la mayor parte, cada mecanismo de interacción en SR tiene su propio conjunto de primitivas RTS. Esto hace al RTS algo largo en términos de tamaño de código, pero permite que cada caracteristica de lenguaje sea implementada bastante eficientemente. Por ejemplo, la implementación de SR de rendezvous es apenas menos eficiente que la mejor implementación Ada de rendezvous.

Se desarrollaron y ejecutaron programas de test para cada mecanismo de interacción entre procesos de SR, y también se midió el efecto del tiempo de context switching. Cada programa de test corrió en una única máquina virtual y ejercitó una única combinación de primitivas. Por ejemplo, el siguiente programa es el usado para determinar el costo de un rendezvous sin argumentos.

```
resource rendezvous( )
  const TRIALS := 10
  var num : int
  op dummy( )
  initial
    getarg(1,num)    # lee argumentos de la línea de comandos
  end
  process server
    do true →
      in dummy( ) → skip ni
    od
  end
  process main
    var start, finish : int, total := 0
    fa t := 1 to TRIALS →
      start := age( )
      fa i := 1 to num → call dummy( ) af
      finish := age( )
      write("elapsed time: ", finish-start)
      total := total + (finish - start)
    af
    write("average time per test: ", total/TRIALS
  end
end
```

El código inicial en este programa lee un argumento de la línea de comandos, *num*, que especifica cuantas veces repetir el test, y luego implícitamente crea ambos procesos. El proceso *main* contiene dos loops. El loop interno hace *num* llamados a la operación *dummy*, que es servida por una input statement en el proceso *server*. El loop externo repite el test 10 veces. La función *age* de SR es llamada antes y después del loop interno; retorna el tiempo consumido, en milisegundos, que el programa SR estuvo ejecutando.

Interpretación de resultados

La siguiente tabla lista el tiempo de ejecución, en microsegundos, de distintas combinaciones de mecanismos de interacción entre procesos. Cada programa de test fue ejecutado varias veces sobre una Sun SPARCstation 1+. El argumento *num* varió desde 10000 hasta un millón para asegurar que los resultados fueran consistentes. Los números mostrados son los valores medios; estos valores fueron producidos por más del 75% de los tests. Los números incluyen overhead de loop, que fue medido como 1.1 microsegundo por iteración:

llamado local	2
par semáforo	5
llamado interrecurso	75
send/receive asincrónico	105
context switch, semáforos	130
context switch, MP	180
send/receive sincrónico	209
creación/destrucción de procesos	350
rendezvous	470
RPC con nuevo proceso	700

Un llamado local es muy rápido. SIn embaro, no es tan rápido como un procedure call C nulo pues un llamado SR contiene 4 argumentos ocultos que se usan para direccionar variables globales y para manejar el caso general de que un llamado podría ser remoto.

El programa de test para semáfos repitió operaciones **V** y **P**, en ese orden de modo que el proceso principal nunca se bloqueaba. Como resultado, las operaciones semáforo son muy rápidas.

El compilador SR optimiza los llamados a procedures en el mismo recurso. Dado que compila los recursos separadamente, sin embargo, no puede optimizar llamados interrecurso. De hecho, el compilador no puede en general saber si una operación en otro recurso es servida por un procedure o por rendezvous o si una instancia de otro recurso estará en el mismo espacio de direcciones. Así, un llamado a una operación no local tiene que ser manejado por el RTS. Si éste determina que la operación es servida por un procedure y está en el mismo espacio de direcciones, entonces el RTS solo hace un jump al procedure. Un llamado interrecurso toma 75 microsegundos debido a este overhead de entrar al RTS y a la necesidad de alocar almacenamiento para los argumentos.

AMP es mucho más costoso que los semáforos pues el RTS tiene que manejar colas de mensajes. También es más costoso que un llamado a procedure interrecurso pues el mensaje tiene que ser copiado en el espacio de direcciones del receptor. El tiempo mostrado en la tabla es el costo de un par de operaciones **send** y **receive**, sin incluir tiempo potencial de context switching.

Los siguientes dos tiempos listados son los costos de context switches. Fueron medidos usando dos programas que forzaban un context switch en cada punto de sincronización. Un programa usó semáforos; el otro AMP. (Los tiempos listados son tiempo total de ejecución menos tiempo de sincronización). El context switching es caro en esta arquitectura pues muchos registrros deben salvarse. En la implementación SR, el context switching es más rápido con semáforos que con MP pues debe salvarse menos información de estado.

Con AMP, en el mejor caso ningún proceso se bloqueará, y por lo tanto no habrá context switch. Con SMP al menos uno de los procesos tiene que bloquearse. Entonces el costo de SMP es el costo de un par **send/receive** más un context switch general.

El tiempo de creación/destrucción de procesos es el tiempo que toma al RTS hacer fork de un nuevo proceso más el tiempo que toma destruir ese proceso cuando termina. El tiempo listado no incluye el overhead de context switching.

El rendezvous es mucho más costoso que el MP. Esto es porque la comunicación es two-way, y porque debe haber dos context switch para completar un rendezvous. El tiempo medido para rendezvous fue 470 microsegundos. Esto es el tiempo para SMP más un gegundo context switch general. El tiempo también es la suma de dos intercambios **send**/**receive** más dos context switches de semáforos (aunque de hecho el llamador no ejecuta realmente **send** y luego **receive** y el tiempo de context switch es algo mayor).

En SR, el mecanismo de interacción más caro es RPC siempre que deba crearse un nuevo proceso para servir el llamado. El tiempo medido fue de 700 microsegundos. Esto es aproximadamente el costo de crear y destruir un proceso más dos context switches generales; de hecho, el tiempo de context switch es algo menor, pero hay overhead para retornar parámetros resultado al llamador.

SR no soporta directamente CCRs o monitores. Sin embargo, pueden ser simulados por procedures y semáforos, de modo que podemos aproximar su performance a partir de los costos de operaciones semáforo y context switches de semáforos.

Estos programas de test fueron probados sobre otras arquitecturas, incluyendo DEC Vaxes, una DECstation 3100, Sun 2 y 3, workstations HP, una workstation Silicon Graphics, y multiprocesadores Encore y Sequent. Naaaturalmente, la performance es mejor en máquinas más rápidas. Sin embargo, las diferencias relativas son las mismas. Además, las tasas son aproximadamente las mismas.

También se corrieron tests de MP en una red local, con el proceso principal ejecutando en una workstation y el proceso server en otro. Cuando el programa de test ejecuta en UNIX, la comunicación es de un orden de magnitud más lento; por ejemplo, RPC toma alrededor de 6.4 milisegundos en lugar de 700 microsegundos. Sin embargo, también hay una implementación stand-alone de SR que emplea un kernel de comunicaciones a medida. En este caso, el RPC remoto toma solo alrededor de 1.3 milisegundos.

Para resumir, usar variables compartidas es siempre menos caro que usar MP. Esto es poco sorprendente pues toma menos tiempo escribir en variables compartidas que alocar un buffer de mensajes, llenarlo, y pasarlo a otro proceso. Así, en una máquina de memoria compartida, el programador debería no usar MP a menos que sea la manera más natural de estructurar la solución a un problema.

En una arquitectura de memoria distribuida, tiene que usarse alguna forma de MP, explícita o implícita. En este caso, si el flujo de información es one-way (por ejemplo, en redes de filtros) entonces el mecanismo más eficiente es AMP. También AMP es el más eficiente y el más fácil de usar para peers interactuantes (por ejemplo en algoritmos heartbeat). Sin embargo, para interacción cliente/servidor rendezvous es apenas menos caro que usar dos pares **send/receive** explícitos. En este caso, rendezvous y RPC también son más convenientes para usar. Más aún, RPC puede ser (y ha sido) implementado bastante eficientemente empleando pools de procesos precreados y buffers de mensajes prealocados. RPC también tiene la ventaja de ser fácil para el programador aprenderlo pues es una extensión directa del llamado a procedure convencional.