96

Concurrente/

Repartido

Paradigma de computación

Andrew P. Bernat

*Asociación de investigación informática*

Patricia J. Teller

*Universidad de Texas en El Paso*

96.1 Introducción

96.2 Arquitecturas de hardware

96.3 Arquitecturas de software

Espera ocupada: concurrencia sin abstracciones • Semáforos

• Monitores • Paso de mensajes

96.4 Sistemas distribuidos

96.5 Enfoques formales

96.6 Idiomas existentes con características de concurrencia

96.7 Cuestiones de investigación

96.8 Resumen

**96.1 Introducción**

La computación concurrente es el uso de múltiples procesos o tareas que se ejecutan simultáneamente para computar una respuesta o resolver un problema. La motivación original para el desarrollo de la computación concurrente fueron las técnicas para compartir múltiples usuarios o trabajos en una sola computadora. Estaciones de trabajo modernas usan este enfoque de manera sustancial. Otra ventaja de la computación concurrente, y la razón para gran parte de la atención actual al tema, es obviamente que resolver un problema usando múltiples computadoras es más rápido que usar solo una. Del mismo modo, existe un poderoso argumento económico para usar múltiples computadoras de bajo costo para resolver un problema que normalmente requiere una supercomputadora costosa. Además, el uso de varias computadoras puede proporcionar tolerancia a fallas.

Además, hay un argumento poderoso adicional para la computación concurrente: el mundo es inherentemente concurrente. Así como cada uno de nosotros participa en una gran cantidad de tareas simultáneas (escuchar mientras se ve mientras se lee, etc.), los sistemas operativos necesitan manejar múltiples tareas que se ejecutan simultáneamente; los robots necesitan participar en una multiplicidad de acciones; los sistemas de bases de datos deben manejar simultáneamente grandes cantidades de usuarios que acceden y actualizan información; A menudo, dividir un problema en tareas concurrentes proporciona una solución más simple y directa.

Como ejemplo, considere el problema de Conway: la entrada es en forma de registros de 80 caracteres (imágenes de tarjetas

en el problema original, que da una idea de cuánto tiempo ha estado presente); la salida debe ser en forma de registros de 120 caracteres; cada par de signos de dólar, '$$', debe ser reemplazado por un solo signo de dólar, '$'; y un espacio ' ', debe agregarse al final de cada registro de entrada. En principio, se puede desarrollar una solución secuencial, pero las complicaciones introducidas requieren manipulaciones de búfer complejas y no obvias. Por otra parte, una solución concurrente que consta de tres procesos es más simple y más elegante. Los tres procesos ejecutan dentro de bucles infinitos las siguientes acciones:

1. Process1 lee registros de 80 caracteres en un búfer de 81 caracteres, coloca un carácter de espacio en la ubicación 81, y luego genera caracteres individuales del búfer secuencialmente.
2. Process2 lee caracteres individuales y los copia en la salida, pero usa una máquina de estado simple para sustituir un solo '$' por dos '$$' consecutivos.
3. Process3 lee caracteres individuales, los guarda en un búfer y genera registros de 120 caracteres.

Para desarrollar una solución implementable, necesitamos decidir cómo los procesos de ejecución independiente se deben comunicar. Un enfoque simple y ampliamente utilizado es agregar dos buffers: Buffer1 almacena caracteres de salida de Process1 para ser ingresado al Process2; Buffer2 almacena los caracteres de salida de Process2 para ingresarlos a Process3.

Para simplificar, suponga que Buffer1 y Buffer2 tienen cada uno un carácter. Así:

1. Process1 lee registros de 80 caracteres en un búfer interno de 81 caracteres, coloca un carácter de espacio en la ubicación 81, y coloca secuencialmente en el Buffer1 caracteres individuales del buffer interno.
2. Process2 lee caracteres individuales de Buffer1 y los coloca en Buffer2, pero usa una máquina de estado simple para sustituir un solo '$' por dos '$$' consecutivos.
3. Process3 lee caracteres individuales de Buffer2, los guarda en un buffer interno de 120 caracteres y produce registros de 120 caracteres.

Esta solución demuestra la esencia del paradigma concurrente: procesos secuenciales individuales que cooperan para resolver un problema. La concurrencia ejemplificada es concurrencia canalizada, donde la entrada de todos los procesos, menos el primero es proporcionado por otro proceso. La cooperación, en este y todos los demás casos, requiere que los procesos:

1. Compartan información y recursos.
2. No interfieran durante el acceso a información o recursos compartidos.

En la solución de Conway, la información se comparte fácilmente a través de los buffers. El principal problema es asegurar

que los accesos concurrentes a los dos buffers no entran en conflicto; por ejemplo, Process2 no intenta recuperar un carácter de Buffer1 antes de que Process1 lo haya colocado allí (lo que llevaría a caracteres basura) y Process1 no intenta colocar un carácter en Buffer1 antes de que el carácter anterior haya sido recuperado por Process2 (lo que llevaría a la pérdida de caracteres).

Un ejemplo más simple de interferencia es proporcionado por el siguiente programa simple (donde las declaraciones dentro del par cobegin-coend se ejecutarán simultáneamente):

x: = 0

cobegin

x: = x + 1

x: = x + 2

coend

Considere el valor de x al final de la ejecución. Porque cada declaración de asignación es en realidad una secuencia de instrucciones a nivel de máquina, varios entrelazados de la ejecución de estas instrucciones resultan en diferentes valores finales para x (es decir, 1, 2 o 3). ¡Claramente, esto es inaceptable!

En cada uno de estos ejemplos, está claro que hay *regiones críticas* en las que dos (o más) procesos tienen secciones de código que no pueden ejecutarse simultáneamente; debemos tener *exclusión mutua* entre las regiones críticas. En el ejemplo de Conway, las regiones críticas incluyen:

* Process1 colocando un valor en Buffer1
* Process2 recuperando un valor de Buffer1
* Process2 colocando un valor en Buffer2
* Process3 recuperando un valor de Buffer2

En el ejemplo simple anterior, cada una de las dos declaraciones de asignación son regiones críticas. La esencia de evitar interferencias es descubrir las regiones críticas y aislarlas. Este aislamiento toma la forma de un "protocolo de entrada" para anunciar la entrada en una región crítica y un "protocolo de salida" para anunciar que la ejecución de la región crítica se ha completado (debajo del '#' se introduce un comentario y el '...' representa el código del programa apropiado):

# protocolo de entrada

...

# código de región crítica

...

# protocolo de salida

...

Este es el modelo básico utilizado por los enfoques de *espera ocupada* y *semáforo* (discutido a continuación). Es un nivel de bajo modelado en el sentido de que se debe prestar especial atención a la ubicación de los protocolos de entrada y salida para asegurar que las regiones críticas estén debidamente protegidas.

Existen otros enfoques de implementación de la concurrencia que resuelven el problema crítico de la región al prohibir cualquier interferencia directa entre procesos concurrentes. Esto se hace al no permitir compartir variables. El enfoque de *monitor* coloca todas las variables compartidas y otros recursos bajo el control de un módulo de monitor único, al que se accede mediante un solo proceso a la vez. El enfoque de *pasar mensajes* es compartir información solo a través de mensajes pasados ​​de proceso a proceso. Ambos enfoques se discuten en este capítulo.

Además de evitar la interferencia en el acceso a los datos, debemos evitar la interferencia en el intercambio de recursos. (por ejemplo, entrada de teclado para múltiples procesos). Además, debemos asegurarnos de que cualquier acción física de procesos de concurrencia, como el movimiento de brazos robóticos, se sincronizan adecuadamente.

Por lo tanto, para desarrollar soluciones concurrentes, requerimos anotaciones para:

1. Especificar qué partes de nuestros procesos pueden ejecutarse simultáneamente
2. Especificar qué información y recursos se compartirán
3. Prevenir la interferencia de procesos concurrentes asegurando la exclusión mutua
4. Sincronizar procesos concurrentes en los puntos apropiados.

Además, cualquier solución propuesta a un problema concurrente debe tener ciertas propiedades (ver, por ejemplo, [Ben-Ari, 1990]):

1. Seguridad: esta propiedad siempre debe ser cierta; Ejemplos incluyen:
2. No interferencia
3. Sin *punto muerto* , que ocurre cuando ningún proceso puede continuar porque todos los procesos están esperando condiciones que nunca pueden ocurrir.
4. Corrección parcial: cuando el programa termina, tiene la respuesta correcta
5. Vitalidad: esta propiedad debe ser cierta eventualmente; Ejemplos incluyen:
6. El programa finaliza (si también tiene la respuesta correcta, esta es la corrección total)
7. *Sin* *carrera*: comportamiento no determinista causado por la ejecución simultánea de procesos
8. *Justicia*: cada proceso tiene una oportunidad de ejecución (esto se ve afectado por la implementación y programación de proceso /hilos)

La verificación o prueba de que las soluciones satisfacen estas propiedades es enormemente complicada por la ejecución concurrente de código: las órdenes particulares de ejecución de código pueden exhibir interferencia o punto muerto mientras que otras proceder bien a la terminación. Volviendo al problema de Conway, suponga la ejecución de Process1 y Process2 se emparejan uniformemente para que Process2 recupere cada carácter colocado por Process1 en Buffer1 antes de que Process1 esté listo para generar otro carácter. En este caso, cuando se prueba, el programa exhibe las propiedades de corrección deseadas, falta de punto muerto, etc. Pero si, debido a una variación en la carga de trabajo del procesador o tipo, Process1 se ejecuta más rápido, luego los caracteres se sobrescribirán y se perderán; por otro lado, si Process2 corre más rápido, los caracteres se repetirán. El hecho de que probamos el programa bajo un conjunto particular de circunstancias (incluso para todas las entradas posibles) es irrelevante para este problema. Pruebas suficientes son imposibles debido a la explosión exponencial en el número de posibles entrelazados de ejecución de instrucciones que pueden ocurrir. El único enfoque totalmente satisfactorio es utilizar métodos formales (técnicas que todavía son predominantemente en desarrollo), que se abordan más adelante en este capítulo.

Este capítulo se centra en las arquitecturas de software utilizadas para la concurrencia, utilizando un conjunto de arquetípicos problemas y sus soluciones para la ilustración. Estos problemas se eligen debido a la frecuencia con que surgen en la informática; El estudio cuidadoso de los problemas reales con frecuencia conduce a la comprensión de que un problema aparentemente complicado es, en el fondo, uno de estos arquetipos. Primero, exploramos brevemente el hardware arquitecturas y su impacto en el software.

**96.2 Arquitecturas de hardware**

El hardware puede influir en la sincronización y la comunicación principalmente con respecto a la eficiencia. *Multiprogramación* es el intercalado de la ejecución de múltiples programas en un procesador; en uniprocesador, un sistema operativo de tiempo compartido implementa multiprogramación. Aunque tal enfoque en un uniprocesador no proporciona la aceleración de ejecución que se discutió en la introducción, sí proporciona la posibilidad de elegancia y simplicidad en la solución del problema, que es el segundo argumento para el paradigma de concurrencia.

Al emplear varias computadoras, tenemos *multiprocesamiento* o procesamiento paralelo. El multiprocesamiento puede involucrar a varias computadoras que trabajan en el mismo programa o en diferentes programas al mismo tiempo. Si se construye un sistema multiprocesador para que los procesadores compartan memoria, entonces los procesos pueden comunicarse a través de variables globales almacenadas en la memoria compartida; de lo contrario, se comunican a través de mensajes pasados ​​de proceso a proceso. A diferencia de un sistema multiprocesador, un sistema distribuido se compone de múltiples computadoras que están alejadas entre sí. Este capítulo se centra en sistemas de multiprogramación y multiprocesamiento con una breve introducción a los problemas adicionales asociados con los sistemas distribuidos.

Además (pero fuera del alcance de este capítulo), una amplia variedad de enfoques híbridos de hardware / software

existe.

**96.3 Arquitecturas de software**

Para especificar una arquitectura de software para implementar la concurrencia, debemos proporcionar la sintaxis y la semántica:

* 1. Especificar qué información y recursos se compartirán
  2. Especificar qué partes de los procesos pueden ejecutarse simultáneamente
  3. Prevenir la interferencia de procesos concurrentes asegurando la exclusión mutua
  4. Sincronizar procesos concurrentes en los puntos apropiados

La primera característica no requiere notación especial (las variables compartidas son simplemente globales), y la tercera y cuarta generalmente se fusionan en uno. Se ha propuesto una gran cantidad de mecanismos de software para soportar estas

caracteristicas; En este capítulo exploramos los más utilizados entre ellos:

1. *Ocupado-espera* : implementable en prácticamente cualquier procesador sin soporte de sistema operativo; esto es concurrencia sin abstracciones

2. *Semáforos* : históricamente el mecanismo satisfactorio más antiguo.

3. *Monitores* : módulos que encapsulan el acceso concurrente a datos compartidos.

4. *Transferencia de mensajes* : una abstracción de nivel superior ampliamente utilizada en sistemas distribuidos

**96.3.1 Espera ocupada: concurrencia sin abstracciones**

Para ilustrar el mecanismo de espera ocupada, utilizamos (siguiendo [Ben-Ari, 1982]), un ejemplo muy simple que consiste de dos procesos concurrentes, cada uno con una sola región crítica. La única suposición hecha es que cada acceso a la memoria es atómico; es decir, procede sin interrupción. Nuestra tarea es garantizar la exclusión mutua; el propósito del ejercicio es demostrar el cuidado con el que se debe diseñar una solución para asegurar propiedades de seguridad y vitalidad discutidas anteriormente.

Nuestro primer enfoque, es asegurar que los procesos, p1 y p2, simplemente se turnen en su regiones críticas

global var turn : = 1

process p1

while true do ->

# región no crítica

...

# protocolo de entrada

while turn = 2 do ->

<nada> #Espera por turno

# región crítica

...

# protocolo de salida

turn: = 2

# resto de cómputo

...

end p1

process p2

while true do ->

# región no crítica

...

# protocolo de entrada

while turn = 1 do ->

<nada> # espera el turno

# región crítica

...

# protocolo de salida

turn: = 1

# resto de cómputo

...

end p2

Este enfoque cumple con las propiedades deseadas pero tiene una falla fundamental: los procesos deben turnarse para ingresar a sus regiones críticas. Si p1 está listo y necesita ejecutar su región crítica a una frecuencia más alta que p2, no puede. Los procesos son un ejemplo de *co-rutinas*, históricamente uno de los primeros enfoques de concurrencia. Si modificamos la solución para permitir que cada proceso proceda a su región crítica si el otro proceso no está en su región crítica, y para luego notificar al otro proceso, obtenemos lo siguiente (donde se usa ci para indicar que pi está en su región crítica):

global var c1: = false, c2: = false

process p1

while true do ->

# región no crítica

...

# protocolo de entrada

while c2 do ->

<nada> # espera el turno

c1: = true # p1 en región crítica

# región crítica

...

# protocolo de salida

c1: = false # p1 fuera de la región crítica

# región no crítica

End p1

process p2

while true do ->

# región no crítica

...

# protocolo de entrada

While c1 do ->

<nada> # espera el turno

c2: = true # p2 en región crítica

# región crítica

...

# protocolo de salida

c2: = false # p2 fuera de la región crítica

# región no crítica

end p2

Ahora, sin embargo, tenemos la posibilidad de que el requisito de exclusión mutua de la región crítica pueda ser violado; es decir, ambos procesos pueden estar en sus regiones críticas al mismo tiempo. (Por ejemplo, supongamos tanto c1 como c2 son falsos; p1 comprueba c2 a través del bucle y decide que puede entrar en su región crítica; antes de establecer c1 en verdadero, p2 comprueba c1 a través de su bucle y decide que puede entrar en su región crítica).

Como se muestra a continuación, esta posibilidad desastrosa se puede eliminar haciendo que un proceso anuncie su intención de ingrese a su región crítica antes de verificar si puede ingresar:

global var c1: = false, c2: = false

process p1

while true do ->

# región no crítica

...

# protocolo de entrada

c1: = true # señal de intención de entrar

while c2 do ->

<nada> # espera el turno

# región crítica

...

# protocolo de salida

c1: = false # p1 fuera de la región crítica

# región no crítica

end p1

process p2

while true do ->

# región no crítica

...

# protocolo de entrada

c2: = true # señal de intención de entrar

while c1 do ->

<nada> # espera el turno

# región crítica

...

# protocolo de salida

c2: = false # p2 fuera de la región crítica

# región no crítica

End p2

Pero ahora hemos planteado la posibilidad de una carrera (cuando p1 establece c1 en verdadero *y* p2 establece c2 en verdadero). Una posible solución a esta dificultad, que aparece a continuación, mueve la declaración de anuncio al bucle, junto con un retraso aleatorio:

global var c1: = false, c2: = false

process p1

while true do ->

# región no crítica

...

# protocolo de entrada

c1: = true # señal de intención de entrar

while c2 do ->

c1: = false # Renunciar a la intención si p2 ya en la región crítica

<retraso>

c1: = true # Intenta de nuevo

# región crítica

...

# protocolo de salida

c1: = false # p1 fuera de la región crítica

# región no crítica

...

end p1

process p2

while true do ->

# región no crítica

...

# protocolo de entrada

c2: = true # señal de intención de entrar

while c1 do ->

c2: = false # Renunciar a la intención si p1 ya en la región crítica

<retraso>

c2: = true # Intenta de nuevo

# región crítica

...

# protocolo de salida

c2: = false # p2 fuera de la región crítica

# región no crítica

...

end p2

Pero esta no es una solución satisfactoria porque exhibe una carrera en la situación (poco probable) de que los dos bucles

procedan en perfecta sincronización. Se puede desarrollar una solución válida, como la que aparece a continuación, volviendo al concepto de tomar turnos cuando corresponda, lo que garantiza la exclusión mutua sin requerir turnos alternos (por lo tanto permitiendo verdadera concurrencia):

global var c1: = false, c2: = false, turn: = 1

process p1

while true do ->

# región no crítica

...

# protocolo de entrada

c1: = true # señal de intención de entrar

turn: = 2 # dar prioridad a p2

while c2 y turn = 2 do ->

<nada> # espera si p2 en región crítica

# región crítica

...

# protocolo de salida

c1: = false # p1 fuera de la región crítica

# región no crítica

...

end p1

process p2

while true do ->

# región no crítica

...

# protocolo de entrada

c2: = true # señal de intención de entrar

turn: = 1 # dar prioridad p1

while c1 y turn = 1 do ->

<nada> # espera si p1 en región crítica

# región crítica

...

# protocolo de salida

c2: = false # p1 fuera de la región crítica

# región no crítica

...

end p2

Esta solución se debe a Peterson [1983]; Dekker presentó la primera solución válida. La importancia del enfoque de espera ocupada es triple:

1. Proporciona una buena introducción a los problemas inherentes al diseño de soluciones concurrentes.
2. Es ejecutable en prácticamente todas las arquitecturas de máquinas sin soporte adicional de software adicional

y es, por lo tanto, adecuado para microcontroladores, etc.

1. Las variantes se usan con frecuencia en implementaciones de hardware.

Sin embargo, este enfoque también tiene dos dificultades:

1. Es muy ineficiente: los ciclos de la máquina se gastan al ejecutar bucles de espera ocupada.
2. La programación a un nivel tan bajo es muy propensa a errores.

**96.3.2 Semáforos**

Dijkstra [1968] presentó el primer mecanismo abstracto para la sincronización en programas concurrentes. Los *semáforo*s, llamado así en relación directa con los semáforos utilizados en las líneas de ferrocarril para controlar el tráfico sobre una pista única, es un tipo de datos abstracto con valor entero no negativo con dos operaciones:

P (s): retraso hasta s> 0, luego s: = s - 1

V (s): s: = s + 1

Cuando un proceso se retrasa en un semáforo, se despierta solo cuando otro proceso ejecuta una operación V en ese semáforo. Por lo tanto, no utiliza ciclos de máquina para verificar si puede continuar. Si hay más de un proceso que se está retrasando en un semáforo, solo uno (el cual depende de la implementación) puede ser despertado por una operación V.

Además, el valor de s se puede establecer en la creación de la instancia a través de la declaración de semáforo; si se establece en 0, entonces algún proceso debe ejecutar la operación V (s) antes de que cualquier proceso ejecute primero la operación P (s) de puede continuar. Con este tipo de datos abstractos, tenemos un mecanismo que maneja interferencias y sincronización.

Notas adicionales:

1. Estas son las dos *únicas* operaciones de sincronización definidas; en particular, el valor de s no es determinable
2. La implementación de estas operaciones debe realizarse en el hardware o en el (no interrumpible) núcleo del sistema.
3. Al dormir mientras espera un semáforo (el retraso en P (s)), un proceso no desperdicia ciclos de máquina por comprobación repetida.
4. Los nombres de las operaciones (P y V) provienen de las palabras holandesas *passeren* (pasar) y *vrygeven* (lanzar); a veces, la señal y la espera se usan en lugar de P y V, respectivamente.
5. Cada una de las operaciones P y V procede atómicamente; es decir, no puede ser interrumpido por otro proceso.

El uso del semáforo en la programación concurrente se relaciona directamente con la analogía del ferrocarril. Cada sección crítica tiene el siguiente aspecto:

global var s : semaphore: = 1

# protocolo de entrada

P(s)

# región crítica

...

# protocolo de salida

V (s)

La inicialización de s a 1 asegura que el primer proceso que ejecuta P (s) continuará. (Surge un punto muerto si s se inicializaran a 0.) Solo el primer proceso para alcanzar su declaración P (s) puede continuar, como procesos posteriores encuentran s = 0 y se retrasan. Cuando el primer proceso finaliza su región crítica, se ejecuta V (s), que establece a 1. Uno de los procesos de espera se despierta, encuentra s> 0, disminuye s y continúa.

Tenga en cuenta la importancia de que estas operaciones sean atómicas; esto asegura que dos procesos no puedan despertarse y cada uno encuentra s> 0.

**96.3.2.1 Semáforos y productor-consumidor**

El problema Productor-Consumidor surge cuando un proceso crea valores para ser utilizados por otro proceso. Ejemplos son el problema de Conway y los buffers de varios tipos, etc. Aquí primero miramos la versión del búfer de elementos múltiples de este problema y luego se agregan múltiples productores y consumidores como refinamiento.

# define el búfer

const N: = ... # Tamaño

var buf [N]: int # buffer

front: = 1 # punteros

rear: = 1

semáforo vacío: = N # cuenta el número de espacios vacíos en el búfer

lleno: = 0 # cuenta el número de artículos en el búfer

process productor

var x: int

while true do ->

# produce x

...

P (vacío) # retraso hasta que haya espacio en el búfer

buf [rear]: = x # pone valor en el buffer

V (lleno) # señal de que el búfer no está vacío

rear: = rear mod N + 1 # actualiza puntero del buffer

end productor

proceso consumidor

var x: int

while true do ->

P (lleno) # retraso hasta que un valor esté en el búfer

x: = buf [frente] # obtiene valor

V (vacío) # señal de que el búfer no está lleno

front: = mod frontal N + 1 # actualización del punto del buffer

# consume x

…

End consumidor

El procesamiento del buffer es convencional; solo el acceso al búfer actual debe colocarse en una región crítica porque no hay posibilidad de interferencia entre las asignaciones de rear y front. Tenga en cuenta también el uso de dos semáforos: vacío para indicar que el productor puede proceder porque hay al menos un espacio vacío en el búfer y lleno para indicar que el consumidor puede continuar porque hay al menos un elemento en el búfer. Aunque es posible resolver este problema con un semáforo, menos resultados de concurrencia. Nota que el semáforo vacío se inicializa a N, el tamaño del búfer. El proceso del productor puede correr hasta N pasos por delante del proceso del consumidor.

Para permitir múltiples productores y / o consumidores, debemos proteger las operaciones del buffer actual con semáforos adicionales para evitar, por ejemplo, que dos productores accedan al rear simultáneamente con operaciones de lectura y asignación. Estos semáforos, mutexR y mutexF, garantizan la exclusión mutua de acceso a los punteros rear y front, respectivamente. No es suficiente usar vacío aquí porque arriba a N los productores podrán continuar a través de la declaración P (vacía).

# define el búfer como anteriormente

semáforo vacío: = N, lleno: = 0

semáforo mutexR: = 1 # exclusión mutua en el puntero rear

mutexF: = 1 # exclusión mutua en el puntero frontal

proceso pi # uno para cada productor

var x: int

while true do ->

# produce x

...

P (vacío) # retraso hasta que haya espacio en el búfer

P (mutexR) # retraso hasta que el puntero rear no esté en uso

# coloca el valor en el búfer y modifica el puntero

buf [rear]: = x; rear: = rear mod N + 1

V (mutexR) # liberar puntero rear

V (lleno) # señal de que el búfer no está vacío

end pi

proceso ci # uno para cada consumidor

var x: int

while true do ->

P (lleno) # retraso hasta que un valor esté en el búfer

P (mutexF) # retraso hasta que el puntero front no esté en uso

# acceder al valor en el búfer y modificar el puntero

x: = buf [frente]; frontal: = mod frontal N + 1

V (mutexF) # liberar puntero front

V (vacío) # señal de que el buffer no está lleno

# consume x

...

fin ci

**96.3.2.2 Semáforos y lectores-escritores**

El modelo de lectores y escritores captura las acciones fundamentales de una base de datos; es decir:

* Sin exclusión entre lectores
* Exclusión entre lectores y escritor
* Exclusión entre escritores

En otras palabras, el software debe garantizar solo una actualización de un registro de base de datos a la vez, y ninguna lectura de ese registro mientras se actualiza.

La solución de semáforo más simple es esperar solo al primer lector; los lectores posteriores no necesitan verificar porque ningún escritor puede escribir si ya hay un lector leyendo (aquí, nr y nw son los números de lectores y escritores activos, respectivamente):

...

nr: = nr + 1

if nr = 1 -> P (nw) # si nadie está leyendo actualmente,

# luego asegúrese de que nadie esté escribiendo

# antes de continuar

# acceso a base de datos

...

nr: = nr - 1

if nr = 0 -> V (nw) # si ya no se estpa leyendo, posiblemente despierte

# escritor, o se prepare para el próximo lector

...

P (rw) # retraso hasta que no haya lectores o escritores

# acceso a base de datos

...

V (rw) # despertar lector o escritor retrasado, o prepararse

# para el próximo lector o escritor

Esta solución da preferencia a los lectores sobre los escritores: los nuevos lectores congelan continuamente a los escritores en espera. Extender esta solución a otros tipos de preferencias, como la preferencia del escritor o el orden de llegada. La preferencia es engorrosa.

Un enfoque más general se conoce como "pasar el testigo"; se extiende fácilmente a otros tipos de preferencias porque el control se entrega explícitamente de un proceso a otro. Aunque una cuidadosa explicación del enfoque no se da aquí, el concepto se resume fácilmente. Un proceso debe verificar para asegurarse de que puede proceder legalmente antes de hacerlo; si no puede continuar, el proceso espera en un semáforo asignado a él.

Por ejemplo, un proceso de escritor verifica si no hay lectores o escritores que se estén ejecutando en la base de datos antes de que proceder; si se están ejecutando en la base de datos, el proceso del escritor duerme, esperando en el semáforo asignado a este. Cuando un proceso termina de acceder a la base de datos, verifica las condiciones y se activa (a través de señalización en el semáforo apropiado) uno de los procesos que esperan la condición. Esta última ópera esencialmente, “pasa el testigo” de un proceso a otro. La clave es que primero se realiza una verificación para garantizar que es legal que el otro proceso se despierte. La fuerza del enfoque de "pasar el testigo" emerge cuando su flexibilidad se utiliza para desarrollar soluciones más generales. Los detalles se pueden encontrar en Andrews [1991].

**96.3.2.3 Dificultades con semáforos en el diseño de software**

Si bien el uso de semáforos proporciona una solución completa al problema de interferencia, la correcta solución depende directamente del uso correcto de las operaciones del semáforo, que son de nivel bastante bajo y desestructurado. Los semáforos y las variables compartidas son globales para todos los procesos y, como cualquier estructura de dato global, su uso correcto requiere una considerable disciplina por parte del programador. Además, si un gran sistema se va a construir, cualquier implementador probablemente sea responsable de solo una parte del uso del semáforo por lo que el emparejamiento correcto de Ps y Vs puede ser difícil. A pesar de esta dificultad, los semáforos son muy utilizados.

**96.3.3 Monitores**

Un enfoque más estructurado es encapsular los datos/recursos compartidos y sus operaciones en un solo módulo llamado *monitor*. Un monitor puede contener datos y procedimientos de acceso no externo que manejan el estado de los recursos. El acceso externo se controla estrictamente a través de llamadas de procedimiento al monitor; se garantiza la exclusión mutua porque la ejecución del procedimiento dentro del monitor no es concurrente.

Los monitores tienen las ventajas tradicionales de los tipos de datos abstractos, pero también deben tratar con dos problemas derivados de su uso mediante la ejecución simultánea de procesos: evitar interferencias y proporcionar sincronización. Esta sección ilustra algunas aplicaciones de muestra de monitores y cómo funcionan internamente.

Volviendo al problema Productor-Consumidor, implementamos un monitor para manejar el acceso compartido al búfer. El monitor requiere un mecanismo de sincronización para garantizar que el productor no pueda sobrellenar el búfer y que el consumidor no puede recuperar de un búfer vacío. Monitores implementan *variables condicionales*, cuyos valores son colas de procesos retrasados ​​en las condiciones correspondientes. Las dos operaciones estándar definidas en la variable condicional cv son:

1. wait (cv): hace que el proceso de ejecución se demore y se coloque al final de la cola de cv; para permitir el eventual despertar del proceso, el proceso debe renunciar al acceso exclusivo al monitor cuando ejecuta una espera.
2. señal (cv): hace que el proceso en la cabecera de la cola del cv se despierte; si la cola está vacía no hay efecto.

Aunque estas operaciones reflejan las de los semáforos, hay una diferencia clave: la operación de la señal no tiene memoria.

**96.3.3.1 Monitores y productor-consumidor**

El monitor de búfer se puede definir de la siguiente manera:

monitor Buffer

# define el búfer

const N: = .. # tamaño del búfer

var buf [N]: int # buffer

front: = 1 # punteros de búfer

rear: = 1

# define las variables de condición

var not\_full, # señalado cuando count <N

not\_empty: cv # señalado cuando count> 0

procedimiento de depósito (datos: int)

if count = N # comprobar espacio

then wait (not\_full) # retraso si no hay espacio

buf [rear]: = datos

rear: = (rear mod N) + 1

N: = N + 1

señal (not\_empty) # señal no vacía

end

procedimiento buscar (datos var: int)

if count = 0 # comprobar si no está vacío

then wait (not\_empty) # retraso si está vacío

datos: = buf [front]

front: = (front mod N) + 1

N: = N - 1

señal (not\_full) # señal no llena

end Buffer

Con este monitor, las tareas de productor y consumidor se pueden rehacer de la siguiente manera:

Process productor

var x: int

while true do ->

# produce x

...

depósito (x)

end productor

proceso consumidor

var x: int

while true do ->

buscar (x)

# consume x

...

End consumidor

Ahora está claro que la programación (fuera del monitor) ahora se puede hacer a un nivel más abstracto, que

conducirá a un software más confiable.

**96.3.3.2 Dificultades con los monitores**

También hay dificultades con los monitores. Considere el caso en el que tenemos dos consumidores, C1 y C2.

Si el búfer está vacío cuando C1 ejecuta buscar, C1 se retrasará si no está vacío. Si el productor entonces ejecuta depósito (tenga en cuenta que depósito y recuperación no se pueden ejecutar simultáneamente), eventualmente señal (no vacía), que despertará a C1. Pero si C2 ejecuta la búsqueda antes de que C1 continúe la ejecución y su llamada a buscar se ejecuta, entonces C1 accederá a un búfer vacío. Por lo tanto, la operación de la señal se considera una *pista de* que es posible proceder con la ejecución, pero no es correcto. Se utilizan dos enfoques para resolver este problema:

1. Reemplace la verificación de la variable de condición con una verificación dentro de un bucle para asegurarse de que la condición es cierto antes de que continúe la ejecución. Por ejemplo:

procedimiento de depósito (datos: int)

while count = N do -> # comprobar espacio

then wait (not\_full) # retraso si no hay espacio

buf [rear]: = datos

rear: = (rear mod N) + 1

N: = N + 1

señal (not\_empty) # señal no vacía

end

1. Dar la máxima prioridad a los procesos de despertar para que el acceso al monitor no sea posible. Esto también requiere que la operación de señal sea la última operación ejecutada en cualquier procedimiento en el que ocurre (para garantizar que dos procesos no se ejecutarán dentro del monitor).

Los monitores forman la base para la programación concurrente en una serie de sistemas y proporcionan un servicio eficiente, mecanismo de sincronización de alto nivel. Tienen la ventaja adicional, al igual que otros tipos de datos abstractos.

u objetos, de permitir modificaciones y ajustes locales sin afectar el resto del sistema.

**96.3.4 Paso del mensaje**

Considere una arquitectura de hardware con múltiples computadoras independientes. Crear un semáforo para el acceso eficiente a los procesos que se ejecutan en computadoras separadas es un problema difícil. Necesitamos una nueva abstracción para este caso: paso de mensajes en el que un proceso de envío envía un mensaje a un *canal* y un proceso de recepción ingresa el mensaje de este mismo canal. Hay una gran cantidad de variaciones de este concepto básico, dependiendo de la semántica de las operaciones y los canales.

Las primitivas básicas son:

1. Declaración del canal
2. enviar <canal> <mensaje>
3. recibir <canal> <variable>

Si los procesos de envío y recepción se bloquean al alcanzar su correspondiente operación de paso de mensaje, tenemos comunicación *sincrónica*; si el proceso de envío puede enviar un mensaje y continuar sin esperar el recibo, el sistema es *asíncrono*. Las analogías son la comunicación telefónica y el sistema postal. El enfoque sincrónico permite la sincronización rápida de procesos (en el instante del paso del mensaje sabemos dónde están ambos en su ejecución). Este fue el enfoque elegido por Hoare [1985] por su modelo de comunicación de procesos secuenciales y su posterior implementación en el lenguaje *occam* [Jones y Goldsmith, 1988]. Si deseamos asincronía, podemos agregar un buffer intermedio de procesos al enfoque sincrónico. Una ventaja de pasar mensajes sincrónicos es que a menudo simplifica el análisis de un algoritmo porque se sabe dónde están los procesos de envío y recepción en su ejecución en el momento en que se pasa el mensaje.

Surgen variaciones adicionales, dependiendo de si los canales son un proceso de un proceso a un proceso o un proceso a muchos procesos, instanciados estáticamente en el momento de la carga o creados dinámicamente durante la ejecución, bidireccionales o unidireccional, si el proceso de recepción debe ser nombrado por el proceso de envío, etc. Sin embargo, el concepto básico es el mismo en todos los casos; La facilidad de uso y la eficiencia de la implementación varían.

Otras variaciones incluyen *llamada a procedimiento remoto* (RPC), que es el núcleo de muchos sistemas distribuidos, y *cita* , el enfoque utilizado en Ada. Exploramos estos enfoques luego de mirar más de cerca al simple mensaje que pasa. Tenga en cuenta que, en el enfoque de paso de mensajes, no hay variables compartidas, por lo que la interferencia no es un

problema. El problema de la sección crítica no surge porque no hay forma de que los procesos concurrentes interfieran juntos. Este es uno de los principales factores de motivación para el uso del software de transmisión de mensajes.

**96.3.4.1 Transferencia de mensajes y productor-consumidor**

Si el sistema de paso de mensajes es asíncrono, como se demuestra a continuación, se puede confiar en los valores que tomará el buffer:

canal P2C

process Productor

int x

while true do ->

# produce x

enviar P2C x

end Productor

proceso consumidor

int x

while true do ->

recibir P2C x

# consume x

end consumidor

Con este enfoque, el productor envía un mensaje a través del canal P2C y continúa produciendo y enviando (hasta la capacidad del canal, en cuyo punto el sistema se bloquea), mientras que el consumidor bloquea la declaración recibir si no hay mensajes disponibles.

Si nuestro sistema es síncrono, como se muestra a continuación, creamos un proceso de almacenamiento intermedio separado:

canal P2B, B2C

Process buffer

# crear el búfer

const N: = ..

var buffer [N]: int

front: = 1

rear: = 1

count: = 0 # número de elementos en el búfer

while true do ->

If

# hay espacio y el productor está enviando

count < n y recibir P2B buffer[rear] ->

count ++; rear: = rear mod n + 1

else

# hay artículos y el consumidor está recibiendo

count> 0 y enviar buffer B2C [front] ->

contar--; front: = front mod n + 1

end Buffer

process Productor

var x: int

while true do ->

# produce x

...

enviar P2B x

end Productor

process consumidor

var x: int

while true do ->

recibir B2C x

# consume x

end consumidor

La declaración if no es determinista; es decir, se puede seleccionar cualquier cláusula verdadera. Las condiciones booleanas en las cláusulas se llaman *guardias* . Las cláusulas son:

* Si hay espacio y el productor desea enviar un caracter
* Si hay artículos para recuperar y el consumidor desea recibir un caracter

Por razones de eficiencia de implementación, los lenguajes de programación reales no permiten guardias para ambos

declaraciones de entrada y salida, por lo que debemos modificar nuestra solución; por ejemplo, como se muestra a continuación, podemos modificar los procesos de almacenamiento intermedio y de consumo para eliminar la protección de la salida:

canal P2B, B2C, C2B

process buffer

# define el búfer

var buffer [n]: int

var front: = 1

rear: = 1

count: = 0

while true do ->

If

# hay espacio y el productor está enviando

count <n y recibir buffer P2B [rear] ->

count ++; rear: = rear mod n + 1

else

# hay artículos y el consumidor solicita

count> 0 y recibir buffer C2B [front] ->

enviar buffer B2C [front]

count--; front: = front mod n + 1

end Buffer

Productor de procesos

var x: int

while true do ->

# produce x

...

enviar P2B x

end Productor

proceso consumidor

var int: x

while true do ->

enviar C2B NIL # anuncia listo para entrada

recibir B2C x

# consume x

...

End consumidor

El proceso del consumidor primero anuncia su intención de recibir un valor del proceso de almacenamiento intermedio (enviar C2B NIL; el NIL significa que no es necesario intercambiar ningún mensaje) y luego recibe el valor (recibir B2C x).

Este programa es un ejemplo de programación *cliente / servidor* . El proceso del consumidor es un cliente del proceso del Buffer; es decir, solicita servicio del búfer, que lo proporciona. La programación cliente/servidor se utiliza ampliamente para proporcionar servicios a través de una red y se basa en el paradigma de paso de mensajes.

**96.3.4.2 Paso de mensajes y lectores-escritores**

El enfoque de pasar mensajes a lectores y escritores es sencillo: no acepte un mensaje de un lector o escritor si un escritor está escribiendo; No acepte un mensaje de un escritor si un lector está leyendo. La solución, que se muestra a continuación, es simple si adoptamos el paso de mensajes sincrónicos y la noción de la base de datos como servidor:

canal Rrequests, Rreceives, Wsends

Lector

enviar solicitudes <mensaje de solicitud>

recibir Rreceives <data>

Escritor

enviar Wsends <escribir mensaje>

Servidor

Si

# no hay escritores, acepta solicitudes de lectores

nw = 0 ->

recibir solicitudes <mensaje de solicitud>

# acceder a la base de datos

...

enviar Rreceives <data>

else

# no hay lectores ni escritores, acepta escritor

peticiones

nr = 0 y nw = 0 ->

recibir Wsends <escribir mensaje>

# modificar la base de datos

...

**96.3.4.3 Paso de mensajes y simulación de semáforo**

Por supuesto, como mostramos a continuación, el paso de mensajes puede simular un semáforo (y viceversa si es necesario):

canales P, V, initSemaphore

proceso semáforo

var s: int

recibir initSemaphore i

s: = i

while true do ->

Si

# el semáforo no es cero acepta la operación P

s> 0 y recibir P NIL->

s--

# siempre acepta la operación V

recibir V NIL ->

s ++

end semáforo

**96.3.4.4 La llamada a procedimiento remoto y las abstracciones de encuentro**

La abstracción de llamada a procedimiento remoto, o RPC, se usa ampliamente para proporcionar servicios cliente / servidor en un sistema distribuido. Revisando los ejemplos de cliente / servidor anteriores, está claro que el cliente ejecuta un par enviar-recibir mientras el servidor ejecuta un par recibir-enviar. Usando el modelo de procedimiento estándar para capturar las acciones del servidor, una declaración de llamada para capturar las acciones del cliente y los parámetros para capturar el mensajes enviados, tenemos:

Cliente

...

llamar al servidor (args)

...

Servidor (formal args)

...

return

que refleja las llamadas a procedimientos tradicionales. La diferencia es que el procedimiento del servidor puede estar en una máquina remota al proceso del cliente. De hecho, el servidor se implementa como un proceso que siempre está retrasado hasta que un cliente ejecuta una llamada. Si varios clientes ejecutan simultáneamente llamadas a un servidor, el servidor debe ser reentrante o debe proporcionar protección para la información compartida. El enfoque RPC forma la base para programas de sistemas distribuidos en una amplia variedad de plataformas; su relación con los monitores debe quedar claro.

El proceso y el procedimiento de llamada no son realmente concurrentes en el sentido utilizado en este capítulo, en que el proceso de llamada se retrasa una vez que se realiza la llamada, el procedimiento no se ejecuta hasta que se llama, el procedimiento se retrasa cuando se ejecuta la devolución y el proceso de llamada reanuda la ejecución solo cuando regreso del procedimiento. El modelo es similar al del mensaje sincrónico que pasa si la ejecución del procedimiento se ve como un componente del proceso de paso de mensajes (esencialmente, el procedimiento crea el mensaje de respuesta).

Podemos aumentar el poder de este enfoque si modificamos el procedimiento en un proceso y tenemos ambos procesos que se ejecutan simultáneamente. Cuando se realiza una llamada, la ejecución de la llamada de procesos demora mientras la ejecución del proceso llamado continúa hasta que esté listo para aceptar la llamada (a través de una declaración especial).

El proceso llamado continúa la ejecución, realiza acciones o calcula valores para el mensaje de retorno. El mensaje de retorno se devuelve a la persona que llama, el proceso llamado continúa ejecutándose y el proceso de llamada reanuda la ejecución una vez que se recibe el mensaje. Porque hay un período de tiempo extendido durante el cual los dos procesos están sincronizados (desde la llamada aceptada hasta la llamada devuelta), este modelo de concurrencia se denomina *cita* . Es la base del modelo de concurrencia utilizado en el lenguaje Ada. El modelo Ada no es simétrico: el proceso de llamada debe conocer el nombre del proceso al que está llamando, pero el proceso llamado no necesita saber quién llama. Las declaraciones de aceptación pueden tener guardias, como se discutió anteriormente para pasar mensajes, para controlar la aceptación de llamadas. La complejidad de estos guardias, y su prioridad, deben ser cuidadosamente seguido durante la implementación del programa.

Este enfoque tiene varias ventajas, todas basadas en la posibilidad de que la rutina llamada utilice múltiples declaraciones de aceptación:

1. La rutina llamada puede proporcionar diferentes respuestas al proceso de llamada en diferentes etapas de su ejecución.
2. La rutina llamada puede responder de manera diferente a diferentes procesos de llamada.
3. La rutina llamada elige cuándo recibirá una llamada.
4. Se pueden usar diferentes declaraciones de aceptación para proporcionar diferentes servicios de manera clara (en lugar de a través de valores de parámetros).

**96.3.4.5 Dificultades con el paso de mensajes**

Los sistemas de paso de mensajes son frecuentemente ineficientes durante la ejecución a menos que se desarrolle el algoritmo cuidadosamente. Esto se debe a que los mensajes tardan en propagarse, y este tiempo es esencialmente una sobrecarga. Por ejemplo, una versión de búfer de un solo elemento del problema de Conway pasa mucho más tiempo intercambiando mensajes que cualquier otra operación.

**96.4 Sistemas distribuidos**

Además de las dificultades inherentes al desarrollo y comprensión de soluciones concurrentes, los sistemas distribuidos contienen el problema fundamental de identificar el estado global. Por ejemplo, ¿cómo determinamos si un programa ha terminado? En el caso secuencial, esto es obvio, ejecutamos la declaración de salida o finalización.

En el caso concurrente, debemos asegurarnos de que todos los procesos estén listos para finalizar. En el caso de la multiprogramación, podemos hacer esto comprobando la cola lista; si está vacío, entonces no hay procesos esperando para ejecutarse, lo que garantiza que nunca se agregará ningún proceso a las colas preparadas (si no se puede ejecutar ningún proceso, entonces no puede haber cambios para crear otro proceso listo). Pero si estamos en un sistema distribuido, no hay una sola cola lista para examinar. Si un proceso está en la cola suspendida en su procesador, puede ser preparado por un mensaje de un proceso en un procesador diferente.

Del mismo modo, aún podemos requerir la exclusión mutua en un recurso del sistema: ¿cómo garantizamos el acceso a través de procesadores? La solución es desarrollar un método para determinar el estado global; ver, por ejemplo, Ben-Ari

[1990].

Si bien aún no ha surgido un verdadero paradigma "distribuido" en el dominio de paradigmas de programación, probablemente evolucionará en el área de los sistemas operativos.

**96.5 Enfoques formales**

Argumentamos anteriormente que la verificación de software en la programación concurrente debe tener en cuenta el mayor número de posibles interacciones entre procesos concurrentes. Obviamente, solo pruebas tradicionales demuestran la presencia de historiales de ejecución "buenos" y no es un mecanismo para verificar ninguna solución: secuencial o concurrente. El uso de una rutina de rastreo para generar historiales de ejecución es una técnica estándar secuencial que se vuelve inviable en el dominio concurrente. Considere, por ejemplo, que *n* procesa cada ejecutar *m* acciones atómicas genera ( *n* \* *m* )! / ( *m* !) *n* historias. Para tres procesos, cada uno ejecutando solo

dos acciones, ¡esto es un total de 90 historias posibles!

La alternativa es utilizar un método formal, matemáticamente riguroso para desarrollar una solución y / o verificar

Una solución completa. Se han aplicado dos enfoques para verificar el software concurrente:

1. Axiomático o asertivo

2. Proceso algebraico

El enfoque axiomático desarrolla aserciones en la lógica del predicado que caracterizan los posibles estados de un cálculo. Las acciones de un programa se ven como transformadores predicados que mueven el cálculo de un estado a otro. El estado inicial se especifica mediante la precondición del cálculo, y el estado final se caracteriza por la condición posterior. Este enfoque ha sido explotado por algún tiempo en el paradigma secuencial; ver Schneider [1997] para una introducción completa al campo en el contexto de concurrencia.

El enfoque algebraico del proceso fue promovido por Hoare [1985], quien también fue pionero en los modelos de concurrencia de grano grueso. El concepto es que las interacciones entre un sistema y su entorno (que son todo lo que finalmente se puede observar) se puede modelar a través de una abstracción matemática llamada proceso (esto es la abstracción del proceso informático como se usó anteriormente). Los procesos se pueden combinar mediante leyes algebraicas para sistemas de forma. La comunicación entre procesos es un ejemplo de esta interacción. Al construir un sistema a través de estas leyes matemáticas y luego transformando las matemáticas abstractas en una implementación lenguaje, uno llega a una solución correcta. El lenguaje *occam* fue diseñado para coincidir con las leyes algebraicas ideado por Hoare; existen transformaciones entre estas leyes y construcciones de programación *occam* (pero las transformaciones no son perfectas debido a los aspectos prácticos de implementación) [Hinchey y Jarvis, 1995]. Varios esfuerzos posteriores desarrollaron álgebras de proceso con propiedades variables [Milner, 1989]; ver Magee y Kramer [1999] por el uso de un álgebra de procesos en el desarrollo de programas Java.

Aunque ambos enfoques están en uso activo, no suelen aplicarse en el paradigma concurrente con mayor frecuencia de la que tienen en el paradigma secuencial, y siguen siendo principalmente herramientas de investigación. La dificultad fundamental es que los teóricos buscan las "partículas fundamentales" de la informática, desarrollar leyes matemáticas que permitan el razonamiento formal. Los lenguajes prácticos son (inherentemente) extremadamente mezclas complejas de estas partículas y leyes fundamentales para tener el poder suficiente para resolver problemas del mundo real. Las herramientas teóricas aún no se adaptan a estos grandes y complejos problemas.

**96.6 Idiomas existentes con características de concurrencia**

Se ha desarrollado una gran cantidad de idiomas para usar el paradigma de concurrencia; la mayoría se han quedado en el ambiente de laboratorio. Si el sistema operativo subyacente proporciona el soporte requerido, entonces los semáforos se pueden implementar en cualquier idioma a través de llamadas al sistema. Estructura de control de concurrencia de nivel superior

Las medidas requieren la modificación del lenguaje secuencial subyacente; por ejemplo, Pascal concurrente [Brinch

Hansen, 1975] usa monitores, mientras que Concurrent C [Gehani y Roome, 1986] se basa en la cita.

Al comenzar con un lenguaje de programación secuencial ampliamente utilizado, un diseñador tiene una gran comunidad

para atraer a los usuarios al nuevo idioma. Ada (concurrencia basada en la cita) y SR

(que incluye estructuras para todos los enfoques discutidos en este capítulo y, por lo tanto, es particularmente

útil para explorar la programación concurrente) [Andrews y Olsson, 1993; ver Hartley, 1995, para una extensa

ejemplos] los idiomas son ejemplos de lenguajes secuenciales con estructuras concurrentes incluidas en

etapas iniciales de desarrollo.

Los lenguajes orientados a objetos también han agregado características de concurrencia. Por ejemplo, Smalltalk tiene

las clases Process y Semaphore para proporcionar la creación dinámica de procesos independientes y sus

interacción utilizando el enfoque de semáforo [Goldberg y Robson, 1989].

Los idiomas basados ​​en un modelo inherentemente concurrente incluyen a Linda (más un idioma independiente

filosofía que un lenguaje) [Ahuja et al., 1986] y *occam* ( *transmisión* sincrónica de mensajes) [Jones y

Orfebre, 1988].

Un enfoque diferente es proporcionar una interfaz estandarizada (una interfaz de programa de aplicación o API)

eso es independiente del lenguaje. Una implementación de lenguaje proporciona un conjunto de rutinas de biblioteca para mejorar

Complementar esta API. Por lo tanto, los programadores pueden usar un lenguaje de su elección mientras se aseguran de que El programa funcionará correctamente. Actualmente, los dos paradigmas principales que son la base para escribir paral-

Los programas lel son mensajes que pasan y memoria compartida. Se utiliza un paradigma híbrido en sistemas compuestos

de nodos multiprocesador de memoria compartida que se comunican mediante el paso de mensajes. Para escribir un mensaje

programas de aprobación, MPI (interfaz de transmisión de mensajes) [ http://www-unix.mcs.anl.gov/mpi/index.html ] es

un estándar ampliamente utilizado; existen muchas variantes de MPI, incluyendo MPICH, CH para Chameleon, que

es una implementación completa y de libre acceso de la especificación MPI, dirigida a un alto rendimiento

[ http://www-unix.mcs.anl.gov/mpi/mpich/ ].

La interfaz de MPI incluye características de varios sistemas de paso de mensajes e intentos de proporcionar

portabilidad y facilidad de uso. El modelo de programación MPI es un MPMD (programa múltiple, datos múltiples)

modelo, en el que cada proceso MPI puede ejecutar un programa diferente. Un cálculo se visualiza como uno

o más procesos que se comunican llamando a las rutinas de la biblioteca para enviar y recibir mensajes a otros

procesos. En general, se crea un conjunto fijo de procesos, uno para cada procesador, en la inicialización del programa

(se anticipan versiones de MPI que admitirán la creación dinámica y la finalización de procesos). Local

y la comunicación global (por ejemplo, difusión y resumen) es proporcionada por punto a punto y colectivo

operaciones de comunicación, respectivamente. El primero se usa para enviar mensajes desde un proceso con nombre

a otro, mientras que este último se utiliza para proporcionar el paso de mensajes entre un grupo de procesos. Más paralelo

Los algoritmos se implementan fácilmente utilizando MPI. Si un algoritmo crea solo una tarea por procesador,

se puede implementar directamente con rutinas de comunicación punto a punto o colectivas que cumplan con su

requisitos de comunicación Por el contrario, si las tareas se crean dinámicamente o si se ejecutan varias tareas

Al mismo tiempo en un procesador, el algoritmo debe ser refinado para permitir una implementación de MPI.

La API de OpenMP se está convirtiendo en un estándar que soporta pro- paralelo paralelo de memoria compartida multiplataforma

gramática en C / C ++ y Fortran en todas las arquitecturas, incluidas las plataformas Unix y Windows NT.

OpenMP es un modelo portátil y escalable que brinda a los programadores paralelos de memoria compartida un sistema simple y

Interfaz flexible para desarrollar aplicaciones paralelas para plataformas que van desde el escritorio hasta el super-

computadora [ http://www.openmp.org/ ]. Esta API está definida conjuntamente por un grupo de hardware informático importante

y vendedores de software. OpenMP se puede usar para dirigir explícitamente la configuración de memoria compartida multiproceso

lelismo Se compone de tres componentes principales de la API: directivas del compilador, rutinas de biblioteca en tiempo de ejecución,

y variables de entorno. Usando el modelo fork / join de ejecución paralela, comienza un programa OpenMP

como un solo hilo maestro. El hilo maestro crea u bifurca un conjunto de hilos paralelos, que concurrentemente

ejecutar una construcción de región paralela. Al finalizar, los hilos se unen a los hilos paralelos (es decir, sincronizar y

terminar), dejando solo el hilo maestro. La API admite paralelismo anidado e hilos dinámicos,

es decir, alternancia dinámica del número de hilos activos. Ámbito variable, por ejemplo, declaración

de datos privados y compartidos, el paralelismo y la sincronización se especifican mediante el uso del compilador

directivas Por sí solo, OpenMP no está diseñado para sistemas paralelos de memoria distribuida. Por ejemplo, para alta-

arquitecturas de clúster de rendimiento como IBM SP, donde se logra la comunicación intranodo

a través de la memoria compartida y la comunicación internado se realiza a través de la transmisión de mensajes, se utiliza OpenMP

dentro de un nodo mientras se usa MPI entre nodos.

Hay muchas herramientas de programación paralela disponibles que ayudan al usuario a paralelizar su aplicación.

y luego portarlo fácilmente a una máquina paralela. Estas máquinas pueden ser máquinas de memoria compartida o una red

de estaciones de trabajo.

**96.7 Cuestiones de investigación**

Si bien está claro que la concurrencia es una técnica necesaria para la solución de muchos problemas, también está claro

ese progreso debe hacerse para asegurar su aplicación efectiva. Que esto sigue siendo un problema de investigación

está claro cuando un sistema operativo falla debido a procesos del sistema que interfieren entre sí o

descubrimos a alguien en nuestro asiento de avión debido al acceso simultáneo a la base de datos de la aerolínea. Esto requiere

El progreso se divide en tres categorías:

1. Se deben hacer avances teóricos para desarrollar técnicas formales que se adapten a las aplicaciones del mundo real.

cationes Por ejemplo, existen verificadores de interferencia de proceso, pero funcionan esencialmente verificando todos

posibles interacciones entre procesos para verificar el punto muerto, etc. Este enfoque se desarrolla rápidamente

Explosión combinatoria.

2. Herramientas de diseño que brinden soporte de desarrollo para soluciones concurrentes. Por ejemplo, depuradores

que capturan el cómputo concurrente sin abrumar al usuario con información.

3. Idiomas con estructuras poderosas para soportar la correcta aplicación de concurrencia. Por ejemplo,

El desarrollo de lenguajes concurrentes orientados a objetos parece sencillo: simplemente permitir

cada objeto se ejecuta simultáneamente porque cada objeto es lógicamente autónomo. Sin embargo, hay un

Número de problemas que necesitan resolución, incluyendo:

a. No todos los objetos deben ejecutarse simultáneamente porque la mayoría de los cálculos seguirán siendo

secuencial (por lo tanto no incurrir en gastos generales del planificador).

si. Si consideramos que varios objetos concurrentes intentan comunicarse con el mismo objeto:

yo. La aceptación de un mensaje debe retrasar todos los demás mensajes para preservar correctamente

estado interno del objeto.

ii) El pedido de aceptación de mensajes debe sincronizarse para garantizar que los cálculos sean correctos.

iii) La aceptación de mensajes debe ocurrir solo en los puntos apropiados de la ejecución del objeto.

C. La herencia a través de la jerarquía de clases crea problemas porque mezclará esta sincronización

con comportamiento de objeto.