CC++ : C++ Concurrente

Alberto Nava

Enzo Chiariotti

Departamento de Computación y

Tecnología de la Información

Universidad Simón Bolívar

Apartado 89000, Caracas 1080A, Venezuela

Abstract

CC++ es una extensión de C++ que provee facilidades para soportar concurrencia. Se incorpora el tipo prototype cuyas instancias, demoninadas *threads*, sólo pueden ser usadas a través de un conjunto de operaciones o *transacciones* . Los threads controlan la forma en la cual las operaciones aplicadas sobre ellos son aceptadas, y es este el único mecanismo de sincronización provisto dentro del lenguaje. También se extienden las nociones de herencia, "freinds" y visibilidad a los prototipos.

Keywords: C++, Concurrencia, Sincronización, Procesos Livianos, Transacciones

1 Introducción

C++ es uno de los lenguajes orientados por objetos más usados en la actualidad, por ser ampliamente disponible y ser compilable en forma eficiente. Desafortunadamente C++ no provee facilidades para programación concurrente.

Se han realizado algunas librerías en C++[3][2] para facilitar este tipo de programación, pero éstas realmente no la soportan en forma completa. También se han desarrollado algunas variantes de C++ que incorporan algunas ideas de concurrencia. cc++[1] incorpora las nociones de "threads" (procesos livianos) ejecutando sobre objetos que equivalen a espacios direccionables distribuidos. Rechazamos cc++ debido a que la granularidad de los objetos no es la más conveniente para algunas aplicaciones, y los threads están soportados en forma completamente diferente que los objetos. C Concurrente[4] (CC) permite compilar programas

LOO[1]	CC++
Objeto	Thread
Clase	Prototipo
Método	Transacción

Figure 1: Esquema Comparativo

en C++, con lo cual se puede utilizar simultáneamente las técnicas orientas por objetos y el soporte a concurrencia por parte del lenguaje. Sin embargo, CC mantiene estos dos mundos separados, lo cual dificulta el desarrollo de muchos programas.

Para añadir concurrencia a C++ nosotros diseñamos un grupo de extensiones que denominamos CC++. Nuestra idea fue extender C++, de forma tal, que un programador entrenado en él pudiese utilizar nuestras extensiones sin mucho esfuerzo.

2 El Lenguaje

En los lenguajes orientados por objetos los objetos son entes pásivos, generados como instancias de clases las cuales se relacionan mediante la herencia. Los objetos son manipulados a través de un conjunto de operaciones definidas en las clases. En CC++ además tenemos los *threads*, como entes activos, que son instancias de *prototipos* los cuales también se relacionan a través de la herencia.

Un thread solo puede ser accedido a través de un conjunto de transacciones definidas en su prototipo. Cuando decimos que un objeto es pasivo nos referimos a que es el llamador el que ejecuta la operación invocada, mientras que en el caso de los threads son éstos los que ejecutan las operaciones sobre ellos invocadas.

[1] Lenguajes Orientados por Objetos

Por ejemplo, para definir un *pipe* síncrono tenemos el siguiente prototipo:

```
prototype Pipe {
public:
    trans void Put(char);
    trans char Take();
private:
    int Inc(int&);
    char *buf;
    int count,size,in,out;
};
```

donde la implementación del *pipe* se basa en un arreglo de tamaño fijo.

Los threads controlan la forma en la cual las operaciones aplicadas sobre ellos son aceptadas, y es éste el único mecanismo de sincronización provisto dentro del lenguaje. Desde el punto de vista del usuario de un thread, una transacción es igual a un función miembro. Pero desde el punto de vista del thread, una transacción es una función miembro sobre la cual se controla su activación. El mecanismo utilizado está inspirado en el rendezvous de ADA, y se extiende con construcciones del tipo *suchthat* y *by* de C Concurrente[4].

El cuerpo del Pipe sería:

```
Pipe::Body()
{
    for(;;)
        select {
            (cout < size) : accept Put();
        or
            (cout > 0)    : accept Take();
        }
}
```

y sus operaciones serían:

```
char Pipe::Take()
{   ++count;return buf[Inc(out)]; }
char Pipe::Put(c)
{   --count; buf[Inc(in)]=c;}
```

Note que las operaciones Put y Take no especifican las sincronizaciones; éstas están en el Body del prototipo.

CC++ es ideal para desarrollar servidores. Por ejemplo para implantar un servidor de nfs[6] tendríamos el prototipo `Nfs_Server`.

```
prototype Nfs_Server {
public:
  lookupres Lookup(lookupargs&);
  readres  Read(fhandle file, fileoffset location, unsiged count);
  writeres Write(fhandle file, fileoffset location, nfsdata data);
private:
}
```

Luego para usar el servidor haríamos `nfs.Lookup(...)`, donde `nfs` es una variable global de tipo `Nfs_Server`.

Note que no hace falta un segundo lenguaje para acceder al servidor. Desde el punto de vista del usuario es equivalente invocar una transacción sobre un thread que esté en el mismo programa o no.

3 Herencia entre los Prototipos

Los prototipos se relacionan a través de la herencia. Esto permite reusar código en forma similar a las clases, pero introduce una nueva dimensión en el reuso. Esta dimensión está representada por la reutilización de especificaciones de sincronización. Es decir, es posible heredar de un prototipo la especificación de cómo se deben ejecutar las transacciones.

Este tipo de herencia junto con la posibilidad de redefinir transacciones en los prototipos es un mecanismo poderoso que facilita los procesos de abstracción y desarrollo.

Por ejemplo para independizar al *pipe* de su representación, tendríamos el prototipo `Pipe` que abstrae el concepto, y prototipos derivados de éste que representan distintas implementaciones.

```
prototype Pipe {
public:
    virtual trans void Put(char);
    virtual trans char Take();
protected:
    Body();
    virtual Boolean Full()=0;
    virtual Boolean Empty()=0;
};

Pipe::Body()
{
    for (;;)
      select {
        (!Full())  : accept Put()
       or
        (!Empty()) : accept Take()
      }
}
```

Este prototipo define su interfaz y las sincronizaciónes necesárias. Las operaciones `Put()`, `Take()`, `Full()` y `Empty()` son virtuales. Es decir, pueden y en éste caso deben, ser redefinadas por los subprototipos.

Note que el `Body` permanece constante para las clases derivadas de `Pipe`.

Ni C Concurrente, ni cc++ permiten herencia entre los prototipos; de hecho cc++ no tiene prototipos ni nada similar. Hasta donde tenemos información no existe un lenguaje basado en C++ que soporte herencia entre prototipos.

4 Transacciones Asíncronas

Inicialmente concebimos las transacciones como síncronas. Es bien conocido que de está forma el software resulta más sencillo, y el la mayoría de los casos se puede emular un comportamiento asíncrono utilizando un generador de ticket y transformando la llamada asíncrona en una notificación de arranque y en una posterior solicitud de confirmación de la completitud de la operación[4].

Sin embargo, es posible que una transacción invoque a otra transacción con lo cual el

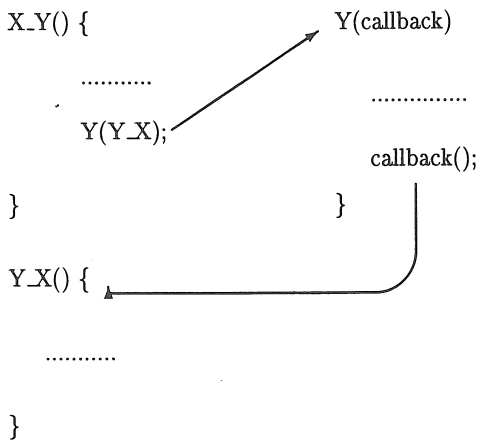

Figure 2:

thread queda impedido de aceptar nuevas transacciones. El problema es similar al de los monitores anidados[5]. Si un servidor invoca una transacción síncrona quedará bloqueado hasta que se complete la transacción, lo cual resulta ineficiente. Por ejemplo el servidor de archivo se bloquearía hasta que se lea un dato del disco, inutilizando por ejemplo el mecanismo de "caching"

Debido a esto incorporamos la noción de llamada a transacción asíncrona, en la cual el llamador no espera por que el llamado acepte y ejecute la transacción. Para especificar que una transacción es asíncrona se le coloca como valor de retorno el token **async**. Por ejemplo

```
prototype Disk_Driver {
public:
   async Request();
}
```

El cuerpo de una transacción X que invoque otra transacción Y queda divide por ésta en dos partes que llamamos X_Y y Y_X. Como muestra la figura 2, la última operación de X_Y es la llamada asíncrona a Y con un nuevo paramétro que es Y_X, de forma tal que al finalizar Y se invoca el forma asíncrona Y_X.

5 Llamadas Anónimas

Otra vez para soportar el desarrollo de servidor introducimos las llamadas a transacciones anónimas, en las cuales no se conoce el nombre del thread llamado. Esto es importante para no ligar sintácticamente los servidor con sus clientes.

Las llamadas anóninas se realizan a través de la noción de apuntador a transacción: Para declarar un apuntador `p` a transación del prototipo `P` que retorna un instancia del tipo `T` hacemos `T (P::*p)()`. Para inicializar `p` hacemos `&a.t()`, donde `a` es una instancia de `P` y `t` es una transacción de `P`. Es decir, `p` representa una instancia y una transacción definida en el prototipo de la instancia en cuestión. Para usar `p` hacemos `(*p)()`. También es posible declarar un apuntador a transacción de cualquier prototipo haciendo T (::*p)(), con lo cual escondemos el tipo de la instancia.

6 Visibilidad de los Miembros de un Prototipo

La visibilidad de la instancias de un prototipo está determinada por la siguiente regla : **una instancia de un prototipo sólo es accesible a través de sus transacciones, esta regla aplica para todos sus clientes. Todo ente distinto de la instancia analizada se considera un cliente de la misma.** Esta regla la llamaremos de integridad.

Un prototipo define un conjunto de transacciones, funciones y objetos. La visibilidad de estas entidades está dada por la aplicación de la regla de integridad a las reglas de visibilidad de C++.

7 Conclusiones

CC++ provee soporte a concurrencia, a través de threads y prototipos. Además permite herencia entre prototipos para facilitar el proceso de desarrollo, y aumentar el reuso. También es posible reusar las especificaciones de sincronización mediante la herencia. CC++ fue implementado sobre el compilador de C++ de GNU lo cual no resultó ser una buena elección, ya que el lenguaje todavía está en una fase de cambios. Actualmente estamos trabajando en un traductor de CC++ a C++.

Referencias

[1] R. Ananthanarayanan. CC++: Preliminary Reference and User Manual. Technical report, Georgia Institute of Technology, Abril 1991.

[2] B. Beck. Shared-Memory Parallel Programming in C++. *IEEE Software*, Julio 1990.

[3] B. Bershad, E. Lazowska, and H. Levy. Presto: A System for Object-Oriented Parallel Programming. Technical report, Computer Science Dept., University of Washington, 1987.

[4] N. Gehani and W. Roome. Concurrent C. *ATT-Bell Laboratories*, Septiembre 1984.

[5] C. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10), Octubre 1974.

[6] R. Sandberg, D. Goldberg, S. Kleiman, S. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Summer Usenix Conference*, 1985.