Software Transactional Memory: a solution to concurrency problems

Duilio J. Protti

FCEIA, Universidad Nacional de Rosario

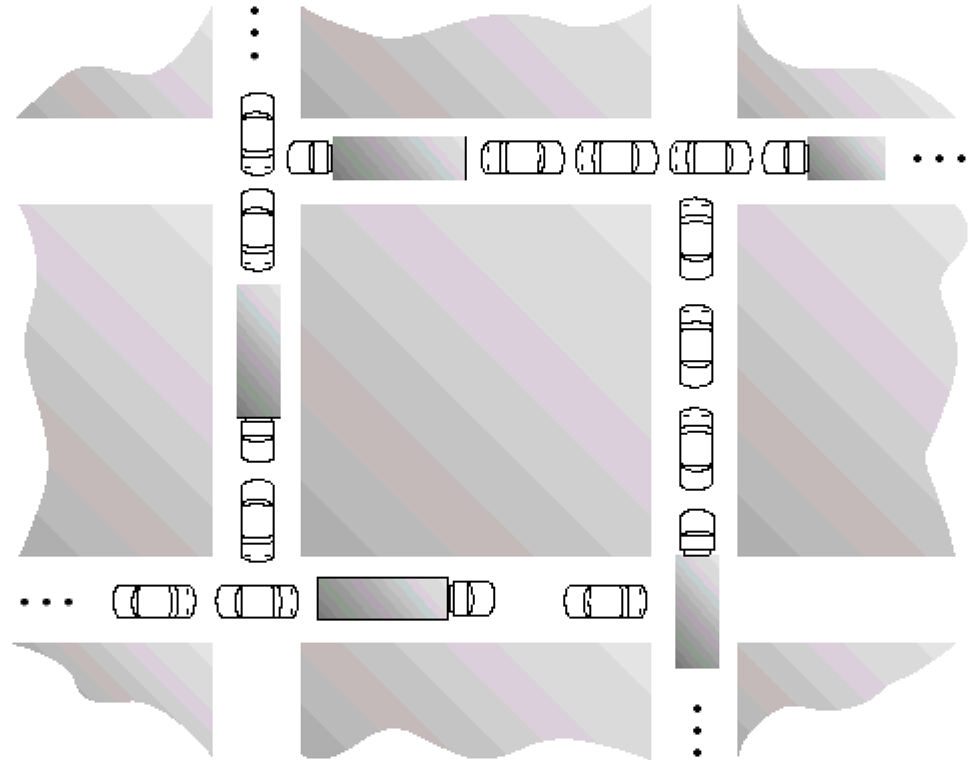
6 de Diciembre de 2005

Deadlock: no es un problema de la programación concurrente, sino de una de sus soluciones

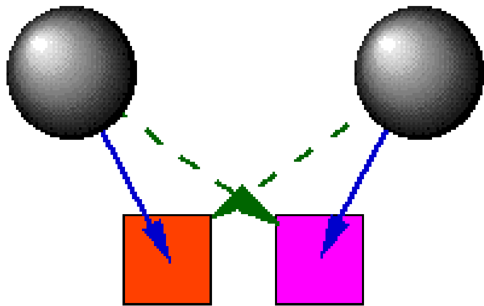
Aparece en los modelos basados en locks

Se puede evitar el deadlock mediante un uso cuidadoso de los locks

O no usándolos...



La opción de no usar locks es la más interesante porque...

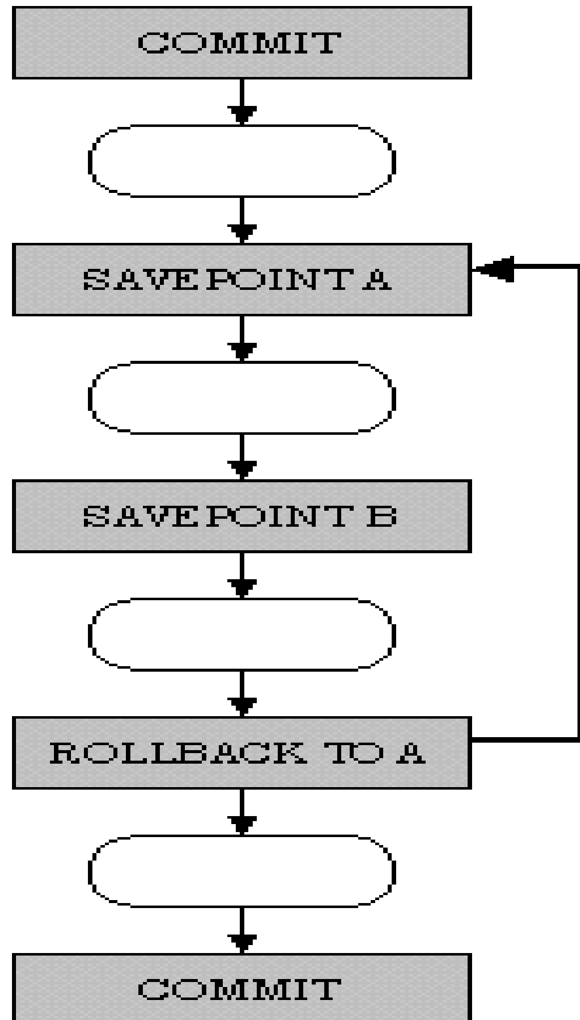


Si no hay locks, no hay deadlock

Pero sin locks ¿se pueden solucionar los problemas de la concurrencia?

Sí, se puede, solo se necesita otro mecanismo

Mecanismo de commit/rollback: se proponen cambios y, si no corresponden, se deshacen



Los participantes no "poseen" un recurso compartido

Cada uno trabaja sobre una copia del recurso compartido

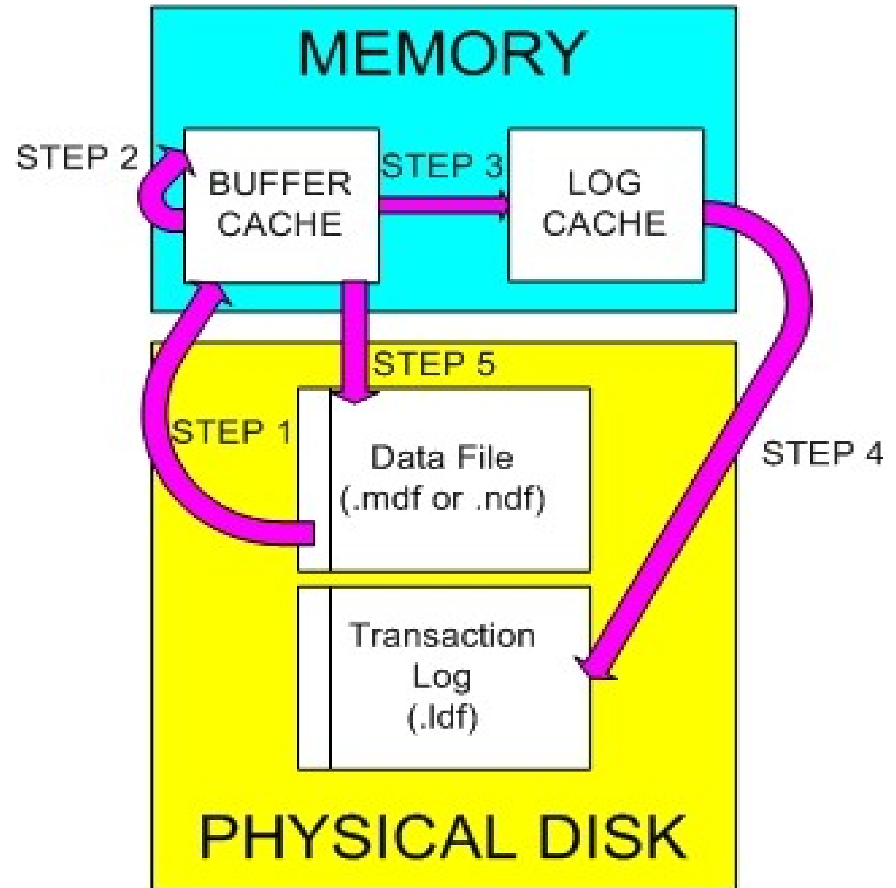
Si al final se vió un estado consistente del recurso, se publican los cambios

Lightweight Memory Transactions: diferentes de las transacciones de las bases de datos

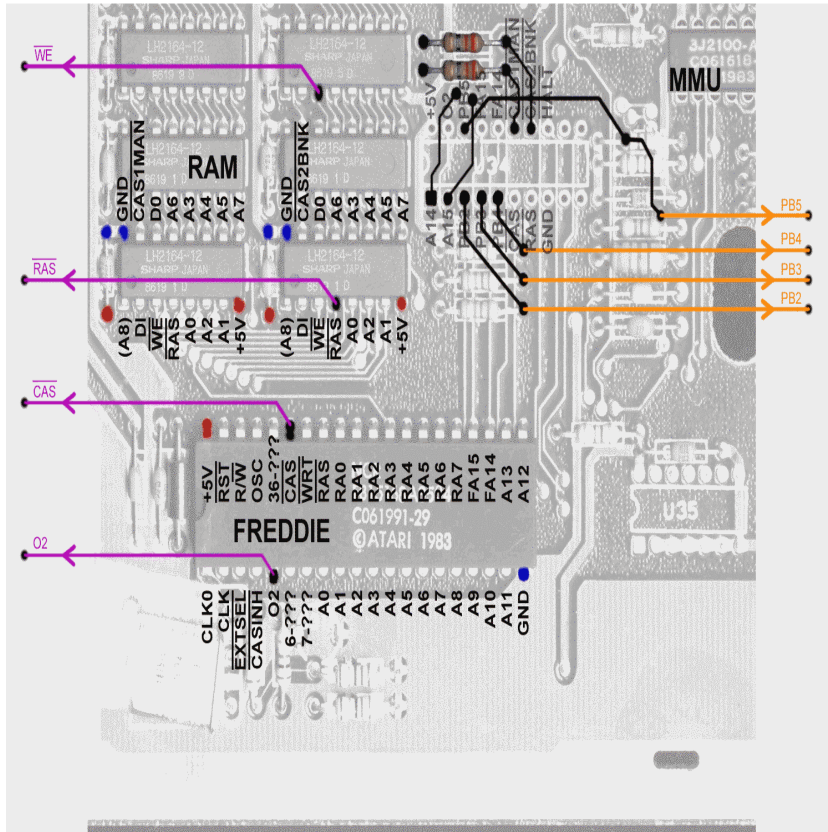
Se toma la *idea* del commit/rollback de las BD

Pero los requerimientos son diferentes

(Persistencia, replicación, tolerancia a fallos, etc)



Software Transactional Memory: transacciones de memoria livianas implementadas en software

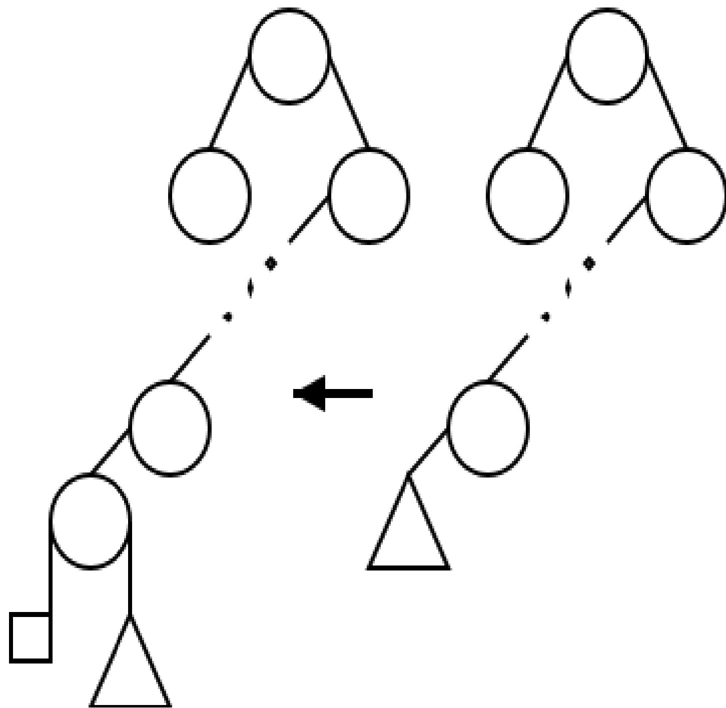


Las transacciones de memoria livianas se propusieron originalmente como una arquitectura de hardware

No triunfaron

Pero con la tecnología actual es factible implementarlas en software

Limitación de implementaciones STM anteriores: falta de componibilidad



Ejemplo: dos árboles, t1 y t2, con operaciones insert() y remove() transaccionales

Se quiere hacer transaccionalmente:

{ x = remove(t1); insert(t2, x) }

Hasta hace poco, esto no escalaba bien

Solución: Composable Memory Transactions, un modelo STM componible

Presentado en el Programming Languages Day de IBM (2005)

Implementado en Haskell (GHC 6.4)

Presenta una forma modular (y componible) de representar acciones atómicas, aún ante la presencia de acciones bloqueantes

Composable Memory Transactions usa un conjunto de combinadores (componibles) de transacciones

```
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()
atomic     :: STM a -> IO a
orElse     :: STM a -> STM a -> STM a
retry      :: STM a
```

atomic permite ejecutar (transaccionalmente) en secuencia dos transacciones

orElse da la posibilidad de ejecutar dos transacciones como alternativas

Antes de ver como trabaja CMT, un repaso a los problemas de los locks

1) Deadlock: debido a locks adquiridos en orden incorrecto.
2) “Wakeups” perdidos: al olvidar notificar a alguna variable de condición.
3) Recuperación de errores muy frágil: liberación de locks en manejadores de excepción.
4) Tensión entre simplicidad y escalabilidad.
5) No son componibles

Ahora veamos cómo trabaja CMT: modelo de concurrencia optimista

atomic

<body>

Se ejecuta <body> sin tomar ningún lock

Cada lectura y escritura en <body> se hace a un log privado al thread en cuestión

En particular, las escrituras van al log, no a la memoria compartida

Al final de la transacción, se intenta "commitear", si no se puede, la transacción se re-ejecuta

Una parte fundamental de CMT: el bloqueo modular

retry() provoca que se abandone el intento actual por ejecutar la transacción, y que se comience desde el principio (aunque no inmediatamente)

El proceso se bloqueará esperando por cualquier cambio en cualquiera de las variables que haya leído hasta el momento

Cuando alguien "commitee" a alguna de esas variables, se despertará

```
atomic
   <... retry; ...>
```

¿Por qué el bloqueo es modular? Por la semántica de la composición orElse

```
atomic
   <<body1>
      `orElse`
     <body2>>
```

Se intenta ejecutar <body1> como transacción, pero si bloquea, se intenta ejecutar <body2> como transacción

Esto permite esperar por varias cosas al mismo tiempo

Es el sueño del select() componible

LibCMT, una implementación de Composable Memory Transactions

- Implementada en C
- Se brindan dos datatypes: GTransaction y GTVar
- Mas los operadores de composición y de ejecución atómica y de bloqueo
- No se requiere garbage collector, ni manejo especial de la memoria, ni un modelo particular de threads
- http://libcmt.sourceforge.net

¿Como usar CMT con LibCMT?

Por cada variable compartida, se crea una variable transaccional (GTVar) que la representa

Cada thread crea las transacciones (GTransaction) que quiere realizar

Las acciones de éstas últimas las realizan funciones que se programan como si el programa fuera secuencial, con la diferencia de que no acceden a variables compartidas, sino transaccionales

Ejemplo simple de uso: incremento de un entero

```
void f (GTransaction *tr, gpointer user_data)
{
    int *i;

    i = g_transaction_read_tvar (tr, tvar_i);
    *i = ++(*i);
}

void worker_thread (void *data)
{
    ...

    tr = g_transaction_new (“Inc”, f, NULL);
    g_transaction_do (tr, NULL);
}
```

Ejemplo simple de composición en secuencia: doble incremento de un entero

```
/* Misma f() que antes */

void worker_thread (void *data)
{
    ...

    tr = g_transaction_new (“Inc”, f, NULL);
    tr2 = g_transaction_sequence (tr, tr);
    g_transaction_do (tr2, NULL);
    ...
}
```

Una nota sobre la implementación del retry: saltos no locales (semi-locales)

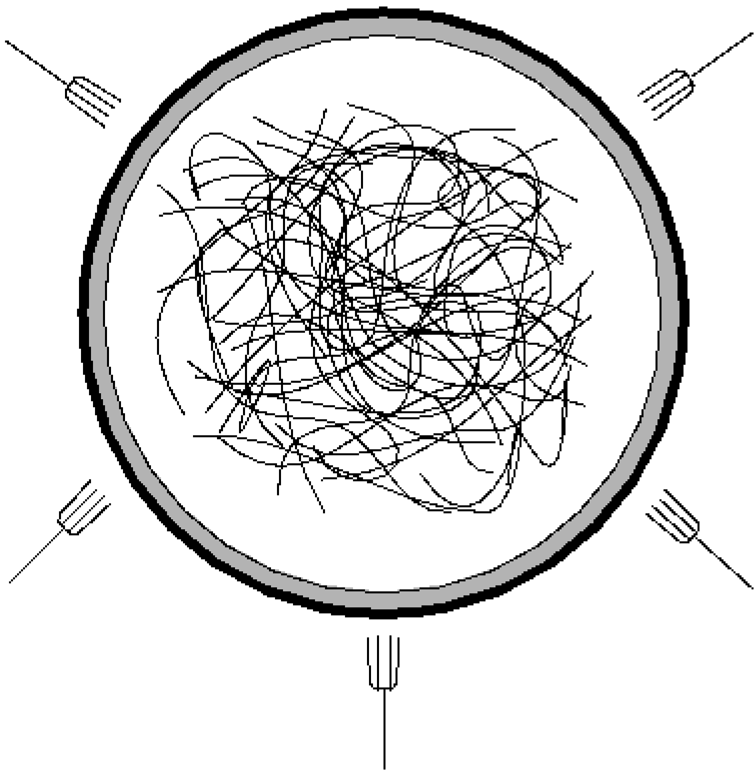
```
void h (GTransaction *tr, gpointer user_data)
{
    int *i;

    i = g_transaction_read_tvar (tr, tvar_i);
    if (*i < N)
        g_transaction_retry (tr);
    ...
}
```

No se dice qué condición se espera, y **no necesariamente se bloquea**

Hacerlo rompería la componibilidad y la facilidad de uso. La biblioteca sabe dónde se debe continuar y a quién se debe notificar los cambios

Ejemplo de composición en alternativas: Problema de los Filósofos Comensales



Se crea una transacción que trata de tomar dos tenedores

Luego se crea take_forks como la composición 'orElse' de las transacciones que tratan de tomar pares de tenedores

Y eso es todo

Composición en alternativas: Problema de los Filósofos

```
void take_pair (GTransaction *tr, gpointer user_data)
{
    fork1 = g_transaction_read_tvar (tr, tforks[index]);
    fork2 = g_transaction_read_tvar (tr, tforks[(index+1)%NP]);
    if (fork1->in_use || fork2->in_use)
        g_transaction_retry (tr);
    fork1->in_use = fork2->in_use = TRUE;
}

void worker_thread (void *data)
{
    take_forks = take_pair_tr[0];
    for (i = 1; i < NP; i++)
        take_forks = g_transaction_or_else (take_forks,
                                            take_pair_tr[i]);
}
```

Volviendo a la STM general, una última ventaja para mencionar: ofrece el mayor paralelismo posible

Un programa concurrente se puede pensar como un orden parcial de sentencias elementales

Dado un programa concurrente, su versión “más paralela” posible es aquella que tiene permitida ejecutar todas las combinaciones posibles dadas por el orden parcial (incluso las incorrectas)

El ideal sería tener las versión “más paralela” posible que sea correcta (desde el punto de vista de la concurrencia)

STM ofrece el mayor paralelismo posible, por diseño

Un mecanismo basado en locks trabaja "serializando" porciones de código (las *critical regions*)

Esto se hace para restringir ejecuciones potencialmente incorrectas. El problema es que se puede restringir más de lo debido. En particular cuando las *critical regions* se anidan (como en transacciones compuestas)

Sin embargo STM permite en principio todas las ejecuciones posibles, luego descarta las incorrectas.

Ofrece el mayor paralelismo, siempre

En resumen, los mecanismos basados en locks no son la única alternativa para tratar la concurrencia

En particular, el deadlock no es inherente a la concurrencia

Moraleja: diferenciar los problemas de un problema, de los problemas de una solución a un problema

Preguntas?