Software Transactional Memory

Juan Quintela – Javier París
{quintela, javier.paris}@udc.es

Monitores

- Estado compartido
- Un conjunto de operaciones atómicas
- Un conjunto de variables de condición
- Un lock implicito

Productores / Consumidores

```
monitor {
  Condition *bufferAvail, *dataAvail;
  int num = 0;
  int data[10];

  Produce(v) {
    while (num == 10) {
      bufferAvail->Wait();
    }
    // put v into data array
    num++;
    dataAvail->Signal();
  }

  Consume(v) {
    while (num == 0) {
      dataAvail->Wait();
    }
    // put next data array value into v
    num--;
    bufferAvail->Signal();
  }
}
```

Productores / Consumidores

```
monitor {
  Condition *bufferAvail, *dataAvail;
  int num = 0;
  int data[10];
  Lock *monitorLock;

  Produce(v) {
    monitorLock->Acquire();
    while (num == 10) {
      bufferAvail->Wait(monitorLock);
    }
    // put v into data array
    num++;
    dataAvail->Signal(monitorLock);
    monitorLock->Release();
  }

  Consume(v) {
    monitorLock->Acquire();
    while (num == 0) {
      dataAvail->Wait(monitorLock);
    }
    // put next data array value into v
    num--;
    bufferAvail->Signal(monitorLock);
    monitorLock->Release();
  }
}
```

Software Transactional Memory

- Optimista
- Marcamos regiones como atómicas
- Se realizan todas las lecturas y escrituras
- Comprueba si algún thread ha cambiado algún valor de los que usamos
- Aborta lo transacción en caso afirmativo
- La termina en caso negativo
- Otro thread ve todos los cambios o ninguno

Sintaxis propuesta

- Insertar en una lista doblemente enlazada

```
// Insert a node into a doubly linked list atomically
 atomic {
     newNode->prev = node;
     newNode->next = node->next;
     node->next->prev = newNode;
     node->next = newNode;
 }
```

STM

- Esperar a que se cumpla una condición

```
atomic (queueSize > 0) {
    remove item from queue and use it
}
```

STM retry

- Espera que un valor leído cambie
- Reintenta

```
atomic {
     if (queueSize > 0) {
         remove item from queue and use it
     } else {
         retry
     }
 }
```

STM

- Tienen que poder componerse

```
get(Key k)  { atomic{seqGet(k)}}
put(Key k, Value v)  { atomic{seqPut(k, v)}}
remove(Key k)  { atomic{seqRemove(k)}}

atomic{ int v = map.get(k);
            v += amount;
            Map.put(k,v);}
```

Read Copy Update

- Alternativa a lock lectores/escritores
- Lecturas tienen muy poco overhead
- Escrituras pueden ser caras
- Mantiene copia de datos antiguos
- Que se reclama cuando todos los lectores antiguos han terminado

Primitivas

```
void rcu_read_lock(void) { }
void rcu_read_unlock(void) { }
void call_rcu(void (*callback) (void *), void *arg)
{
        // add callback/arg pair to a list
}
void synchronize_rcu(void)
{
    int cpu;
    for_each_cpu(cpu)
        schedule_current_task_to(cpu);
    for each entry in the call_rcu list
        entry->callback (entry->arg);
}
```

Primitivas II

```
#define rcu_assign_pointer(p, v) ({ \
                                   smp_wmb(); \
                                   (p) = (v); \
                                 })
#define rcu_dereference_pointer(p) ({ \
           typeof(p) _value = (p);  \
           smp_rmb(); /* not needed on all architectures */ \
           (_value);
         })
```

Comparación lectores/escritores

```
1 struct el {
2   struct list_head lp;
3   long key;
4   spinlock_t mutex;
5   int data;
6   /* Other data fields */
7 };
8 DEFINE_RWLOCK(listmutex);
9 LIST_HEAD(head);
```

```
1 struct el {
2   struct list_head lp;
3   long key;
4   spinlock_t mutex;
5   int data;
6   /* Other data fields */
7 };
8 DEFINE_SPINLOCK(listmutex);
9 LIST_HEAD(head);
```

Comparación II

```
 1 int search(long key, int *result)
 2 {
 3   struct el *p;
 4
 5   read_lock(&listmutex);
 6   list_for_each_entry(p, &head, lp) {
 7     if (p->key == key) {
 8       *result = p->data;
 9       read_unlock(&listmutex);
10       return 1;
11     }
12   }
13   read_unlock(&listmutex);
14   return 0;
15 }
```

```
 1 int search(long key, int *result)
 2 {
 3   struct el *p;
 4
 5   rcu_read_lock();
 6   list_for_each_entry_rcu(p, &head, lp) {
 7     if (p->key == key) {
 8       *result = p->data;
 9       rcu_read_unlock();
10       return 1;
11     }
12   }
13   rcu_read_unlock();
14   return 0;
15 }
```

Comparación III

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   write_lock(&listmutex);
 6   list_for_each_entry(p, &head, lp) {
 7     if (p->key == key) {
 8       list_del(&p->lp);
 9       write_unlock(&listmutex);
10       kfree(p);
11       return 1;
12     }
13   }
14   write_unlock(&listmutex);
15   return 0;
16 }
```

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   spin_lock(&listmutex);
 6   list_for_each_entry(p, &head, lp) {
 7     if (p->key == key) {
 8       list_del_rcu(&p->lp);
 9       spin_unlock(&listmutex);
10       synchronize_rcu();
11       kfree(p);
12       return 1;
13     }
14   }
15   spin_unlock(&listmutex);
16   return 0;
17 }
```