```
/*****************************************************************************
 *
 * Filename:         dbs.cpp
 * Created On:       17.04.1997
 * Created By:       Bettina Kemme
 * Last Modified On:
 * Last Modified By:
 *                                                                           *
 *****************************************************************************
 * Contents: initializing, starting of processes, general management
 *****************************************************************************/

#include "transactMgr.h"
#include "lockMgr.h"

// reading input parameters
#define NUMBER_PARAM_INT 3     // up to now: read in 3 int parameters
#define NUMBER_PARAM_DOUBLE 4  // read in 4 double parameters

int Int_Param_Values[NUMBER_PARAM_INT];
double Double_Param_Values[NUMBER_PARAM_DOUBLE];


int T_SIZE_AVERAGE = 10;            // number of operations per transaction
int T_UPDATE_PERC  = 50;            // percentage of update operation per transaction
int NUMBER_DATA_ITEMS = 100;        // for debugging: make this very small!!!
int LOCK_TABLE_SIZE = 20;           // size of hash array
double TOTAL_SIM_TIME = 100.0 ;     // simulation time
double INTER_ARRIVAL_TIME = 1.0;    // time that passes between arrival of two transact
ions
double DEADLOCK_TIME_OUT = 5.0;     // time after which a waiting lock request leads to
 abort
double OPERATION_TIME = 1.0;        // time of the execution of an operation

Cl_Lock_Table *Lock_Table;          // global pointer to Lock
event *Any_Lock_Granted;            // is set whenever a lock is granted

/* interesting statistics */
table  *MPL = new table("parallel transactions");
table  *RESP = new table("response time");
int transStarted=0,transFinished=0,transCommitted=0,transAborted=0;


/*****************************************************************************
 ****************************** Function: initialize_Params*******************
 *****************************************************************************
 Purpose:    Assigns the values read in the input file "parameter_file" to
             global variables
 *****************************************************************************/
void
initialize_Params()
{
  istream *p_File;
  int i;
  int Number_of_Words;
  String Command_String;
  String Words_List [2];

  p_File = new ifstream("parameter_file");

  for (i=0;i<NUMBER_PARAM_INT+NUMBER_PARAM_DOUBLE;i++)
  {
    if (!p_File->eof())
      readline(*p_File,Command_String);
    else
      Command_String = " ";

    Number_of_Words = split(Command_String,Words_List,2,RXwhite);
```



```
    if (i<NUMBER_PARAM_INT)
      Int_Param_Values [i] = atoi (Words_List [1]);
    else
      Double_Param_Values [i-NUMBER_PARAM_INT] = atof (Words_List [1]);
  };

  T_SIZE_AVERAGE = Int_Param_Values[0];
  T_UPDATE_PERC  = Int_Param_Values[1];
  NUMBER_DATA_ITEMS = Int_Param_Values[2];

  TOTAL_SIM_TIME = Double_Param_Values[0];
  OPERATION_TIME = Double_Param_Values[1];
  INTER_ARRIVAL_TIME = Double_Param_Values[2];
  DEADLOCK_TIME_OUT = Double_Param_Values[3];
}; // initialize params

/****************************************************************************
 ****************************** Function: report ****************************
 *****************************************************************************
 Purpose:    Prints interesting statistics to stdout
 *****************************************************************************/
void
report_statistics()
{
  int i;

  cout << "Ausgangswerte: " << endl;
  cout << "Simulationszeit: " << TOTAL_SIM_TIME << endl;
  cout << "Transaktionsgroesse: " << T_SIZE_AVERAGE << endl;
  cout << "Update-Rate in %: " << T_UPDATE_PERC << endl;
  cout << "Dauer einer Operation: " << OPERATION_TIME << endl;
  cout << "Datenbankgroesse in Objekten: " << NUMBER_DATA_ITEMS << endl;
  cout << "Zwischenankunftsintervall: " << INTER_ARRIVAL_TIME << endl;
  cout << "TimeOut fuer Deadlock: " << DEADLOCK_TIME_OUT << endl << endl;

  cout << "Ergebnisse: " << endl;
  cout << "Transaktionen insgesamt: " << transFinished << endl;
  cout << "Transaktionen commited: " << transCommitted << endl;
  cout << "Transaktionen abgebrochen: " << transAborted << endl;
  cout << "Durchsatz pro Zeiteinheit: " << transCommitted / TOTAL_SIM_TIME << endl;
  cout << "Durchschnitt parallel laufende Transaktionen: " << MPL->mean() << endl;
  cout << "Durchnschnittliche Antwortzeit: " << RESP->mean() << endl;
}; // report_statistics


/****************************************************************************
 ****************************** transaction_Manager ***************************
 *****************************************************************************
 Purpose:    Launches a process that performs exactly one transaction
 *****************************************************************************/
void
transaction_Manager (int transaction_id)
{
  char str[9];
  Cl_Transaction *this_transaction;
  double start_time;

  sprintf (str, "transM#%d", transaction_id);

  // create a new transaction process - csim-feature
  create (str);

#ifdef DEBUG
  cout << "starting transaction " << transaction_id << endl;
#endif

  // statistics
  transStarted++;
```

```
  start_time = simtime();

  this_transaction = new Cl_Transaction(transaction_id);

  this_transaction->Transaction_Loop();
  delete this_transaction;

  // statistics
  transFinished++;
  RESP->record(simtime()-start_time);

  // terminates the process
  terminate();
}; // transaction_Manager


/********************************************************************************
 ****************************** Function: sim *********************************
 *******************************************************************************
 Purpose:    Starts the simulation
             starts the transaction processes with a delay of the interarrival time
 ******************************************************************************/

extern "C" void sim ()
{
  int transaction_id=0;

  set_model_name("Transaction Processing");
  create("sim");

  // initializing
  initialize_Params();
  Any_Lock_Granted = new event;   // csim-feature to initialize signal
  Lock_Table = new Cl_Lock_Table();

  // start new transactions as long as the simulation time has not exceeded
  while (simtime() < TOTAL_SIM_TIME)
  {
    MPL->record(transStarted-transFinished);  // statistic gathering

    // start a new transaction
    transaction_Manager(transaction_id++);

    // wait inter_arrival_time before starting next transaction
    hold(expntl(INTER_ARRIVAL_TIME));
  };
  report_statistics();
  terminate();
};
```

```
/*****************************************************************************
 *
 * Filename:         globals.h
 * Created On:       17.04.1997
 * Created By:       Bettina Kemme
 * Last Modified On:
 * Last Modified By:
 *
 *****************************************************************************
 * Contents:  defines, global variables
 *****************************************************************************/

#ifndef _globals_h
#define _globals_h

#include <String.h>
#include <iostream.h>
#include <fstream.h>
#include <cpp.h>
#include <DLList.h>

#define DEBUG
#undef DEBUG  // if you want DEBUG, then comment this line


#define LOCK_IMPLEMENTED  1      // include once the lock manager is implemented
#define LOCK_GRANTED        100    // lock can be granted immediately
#define LOCK_NOT_GRANTED    101    // lock must wait
extern event *Any_Lock_Granted;    // To signal transactions that have been waiting
                                   // for a lock that a lock has been granted
                                   // event is a csim-feature

extern int T_SIZE_AVERAGE;         // number of operations per transaction
extern int T_UPDATE_PERC;          // percentage of update operation per transaction
extern int NUMBER_DATA_ITEMS;      // for debugging: make this very small!!!
extern int LOCK_TABLE_SIZE;        // size of hash array
extern double TOTAL_SIM_TIME;      // simulation time
extern double INTER_ARRIVAL_TIME;  // time that passes between arrival of two transacti
ons
extern double DEADLOCK_TIME_OUT;   // time after which a waiting lock request leads to
abort
extern double OPERATION_TIME;      // time of the execution of an operation

// forward declaration
class Cl_Lock_Table;
extern Cl_Lock_Table *Lock_Table;  // The Lock Table Entry Point is global

class Cl_Lock_Table_Entry;


// statistics
extern int transCommitted,transAborted;

extern void report_statistics();

#endif _globals_h
```

```
/****************************************************************************
 *
 * Filename:         transactMgr.h
 * Created On:       January 1997
 * Created By:       Stefan Pleisch
 * Rewritten On:     16.04.1997
 * Rewritten By:     Bettina Kemme
 *
 ****************************************************************************
 * Contents:  Description of Transaction and Operations of Txn
 ****************************************************************************/

#ifndef _transactMgr_h
#define _transactMgr_h 1

#include "globals.h"

// Operation can be Read or Write
enum T_Op_Type {
  Read,
  Write
  }; // T_Op_Type

// Operation has not yet started, waits for lock or lock is granted
enum T_Op_State {
  Nothing,
  Lock_Wait,
  Lock_Grant
}; // T_Op_State


// the operation in process
class Cl_Op
{
  T_Op_Type            Op_Type;         // Read or write
  int                  Op_Address;      // the data-item identificator
  T_Op_State           Op_State;        // not yet done, wait for lock, granted lock

  // the next two are used, when this Class is also used as the lock-request chained in
  // the lock hash table
  int                  Transaction_ID;  // backpointing to transaction
  Cl_Lock_Table_Entry  *Lock_Entry_Ptr; // pointer to the lock-entry, it is included

public:
  // Constructor
  Cl_Op (){};
  // Destructor
  ~Cl_Op (){};

  // initialize the operation
  op_init(T_Op_Type op_type, int address, int trans_id);

  // Returns Values
  T_Op_Type get_Op_Type () {return (Op_Type);};
  int get_Op_Address () {return (Op_Address);};
  T_Op_State get_Op_State () {return Op_State;};
  int get_Transaction_ID () {return Transaction_ID;};
  Cl_Lock_Table_Entry* get_Lock_Entry_Ptr () {return Lock_Entry_Ptr;};

  // Settings
  void set_Op_State (T_Op_State state) {Op_State = state;};
  int set_Lock_Entry_Ptr (Cl_Lock_Table_Entry* entry_ptr) { Lock_Entry_Ptr = entry_ptr;
};
}; // Cl_Op


class Cl_Transaction
{
```



```
  int   Transaction_ID;
  Cl_Op *Ops_Array;        // The list of operations belonging to this transaction
  int   Number_of_Ops;     // Total number of operations in this transaction

public:
  // Constructor
  Cl_Transaction (int transaction_id);
  // Destructor
  ~Cl_Transaction(){delete [] Ops_Array;};

  // Return values
  int get_Transaction_ID () {return Transaction_ID;};
  Cl_Op * get_Op(int op_id){return &Ops_Array[op_id];};
  int get_Number_Ops () {return Number_of_Ops;};

  void Transaction_Loop();   // performs all operations of the transaction, request loc
ks
  void Transaction_Commit(); // releases all granted locks and terminates transaction
  void Transaction_Abort();  // releases all waiting and granted locks and terminates t
ransaction
}; // Cl_Transaction

#endif // _transactMgr_h
```

```
/*****************************************************************************
 *
 * Filename:        transacMgr.cpp
 * Created On:      16.04.1997
 * Created By:      Bettina Kemme
 * Last Modified On:
 * Last Modified By:
 *
 *****************************************************************************
 * Contents: implements transaction generation and execution
 *****************************************************************************/

#include "transactMgr.h"
#include "lockMgr.h"

/******************************************************************************
************************************ Cl_Op ***********************************
******************************************************************************/
Cl_Op :: op_init (T_Op_Type op_type, int address, int trans_id)
{
  Op_Type = op_type;
  Op_Address = address;
  Op_State = Nothing;
  Transaction_ID = trans_id;
}; // Cl_Op :: op_init

/******************************************************************************
***********************************Cl_Transaction *************************
*****************************************************************************
*  Purpose: Transaction Generation depending on Input parameters
******************************************************************************/
Cl_Transaction :: Cl_Transaction (int transaction_id)
{
  int i,j,a;
  T_Op_Type op_type;

  Transaction_ID = transaction_id;

  /* size of operation */
  Number_of_Ops = T_SIZE_AVERAGE;

  /* allocate memory for operations */
  Ops_Array = new Cl_Op[Number_of_Ops];

  // create the operations
  for (i=0;i<Number_of_Ops;i++)
  {
    Ops_Array[i].set_Op_State(Nothing);

    // random-function is provided by csim
    a = random(0,100);
    if (a < T_UPDATE_PERC)
      op_type = Write;
    else
      op_type = Read;

    a = random(0,NUMBER_DATA_ITEMS - 1);

    Ops_Array[i].op_init(op_type,a,transaction_id);

    // the only lock combination allowed on a single item
    // is read -> write, the others do not make that much sense!
    for (j=0;j<i;j++)
    {
      if ((Ops_Array[j].get_Op_Address() == a) &&
          ((Ops_Array[j].get_Op_Type() == Write) ||
           (Ops_Array[i].get_Op_Type() == Read)))
      {
```



```
        i--;
        break;
      };
    };
  };
}; // Cl_Transaction :: Cl_Transaction


/******************************************************************************
********************* Cl_Transaction :: Transaction_Commit *******************
******************************************************************************
*  Purpose: release locks
******************************************************************************/
void
Cl_Transaction :: Transaction_Commit()
{
  int i;
  T_Op_State op_state;

  for (i=0;i<Number_of_Ops;i++)
  {
    if (Ops_Array[i].get_Op_State() != Lock_Grant)
    {
      cout << "Commit but not yet all locks granted" << endl;
      report_statistics();
      exit(-1);
    }
    else
      Lock_Table->release_Lock(&Ops_Array[i]);
  };
#ifdef DEBUG
  cout << "transaction committed" << Transaction_ID << endl;
#endif
  transCommitted++;
}; // Cl_Transaction :: Transaction_Commit


/******************************************************************************
********************* Cl_Transaction :: Transaction_Abort *******************
******************************************************************************
*  Purpose: release locks and waiting locks, do not release operations that have
*           not yet requested a lock
******************************************************************************/
void
Cl_Transaction :: Transaction_Abort()
{
  int i;
  T_Op_State op_state;

  for (i=0;i<Number_of_Ops;i++)
  {
    op_state = Ops_Array[i].get_Op_State();
    if (op_state == Nothing)
      break;
    else if (op_state == Lock_Grant)
      Lock_Table->release_Lock(&Ops_Array[i]);
    else
      Lock_Table->remove_Wait(&Ops_Array[i]);
  };
#ifdef DEBUG
  cout << "transaction aborted" << Transaction_ID << endl;
#endif
  transAborted++;
}; // Cl_Transaction :: Transaction_Abort


/******************************************************************************
********************* Cl_Transaction :: Transaction_Loop   *******************
```

```
******************************************************************************
*  Purpose: perform all operations
*****************************************************************************/
void
Cl_Transaction :: Transaction_Loop()
{
  int i,result;

#ifndef LOCK_IMPLEMENTED

  for (i=0;i<Number_of_Ops;i++)
    hold(expntl(OPERATION_TIME));
  transCommitted++;

#else

  for (i=0;i<Number_of_Ops;i++)
  {
    result = Lock_Table->request_Lock(&Ops_Array[i]);
    if (result == LOCK_GRANTED)
       hold(expntl(OPERATION_TIME));
    else
    {
      Transaction_Abort();
      return;
    };
  };
  Transaction_Commit();

#endif
}; // Cl_Transaction :: Transaction_Loop()
```

```
/*****************************************************************************
 *
 * Filename:         lockMgr.h
 * Created On:       January 1997
 * Created By:       Stefan Pleisch
 * Rewritten On:     16.04.1997
 * Rewritten By:     Bettina Kemme
 *
 *****************************************************************************
 * Contents: description of the lock table and the entries
 *****************************************************************************/

#ifndef _lockMgr_h
#define _lockMgr_h 1

#include "globals.h"

// Forward declarations
class Cl_Lock_Table_Entry;
class Cl_Op;


// List of lock table entries
typedef DLList<Cl_Lock_Table_Entry *> T_Lock_List;
// List of operations that have granted locks or wait for locks
typedef DLList<Cl_Op *> T_Op_List;

// one entry in the hash table
class Cl_Lock_Table_Entry
{
  int Address;       // Identification of the data item the lock is for

  T_Op_List *Op_List_Granted;  // List of operations with locks granted
  T_Op_List *Op_List_Waiting;  // List of operations waiting for locks

public:
  // Constructor
  Cl_Lock_Table_Entry (int address);
  // Destructor
  ~Cl_Lock_Table_Entry ();

  // Returns the address of the data item the lock is protecting
  int get_Address () {return Address;};

  // tries to acquire lock, includes op in the Lock Table
  int request_Lock(Cl_Op *op);
  // removes op from the lock table and grants new locks
  void release_Lock(Cl_Op *op);
  // removes waiting request from the lock table
  void remove_Wait(Cl_Op *op);

  // Returns 1 if all lists of this Lock_Table_Entry are empty, i.e. no
  // ..operation is waiting for this lock any more. 0 otherwise.
  int unused ();
}; // Cl_Lock_Table_Entry

// the lock table
class Cl_Lock_Table {

  // Hash table to improve performance
  T_Lock_List **Hash_Table;

  // Hashes the address of a data item and returns the index in the hash table
  int hash_Address (int Address) {return (Address%LOCK_TABLE_SIZE);};

  // returns the entry for a certain address
  // if the entry does not exist, it creates it
  Cl_Lock_Table_Entry *find_Entry(int address);
```



```
  // removes an entry
  void remove_Entry(Cl_Lock_Table_Entry *entry);

public:
  // Constructor
  Cl_Lock_Table ();
  // Destructor
  ~Cl_Lock_Table ();

  // looks for the right entry and requests lock
  int request_Lock (Cl_Op *op);
  // looks for the right entry and releases lock
  void release_Lock(Cl_Op *op);
  // looks for the right entry and releases lock-request
  void remove_Wait(Cl_Op *op);
}; // Cl_Lock_Table

#endif // _lockMgr_h
```

```
/******************************************************************************
 *
 * Filename:         lockMgr.cpp
 * Created On:       16.04.1997
 * Created By:       Bettina Kemme
 * Last Modified On:
 * Last Modified By:
 *
 ******************************************************************************
 * Contents: implements the lock requests and releases
 ******************************************************************************/

#include "lockMgr.h"
#include "transactMgr.h"

/*******************************************************************************
 **************************** Cl_Lock_Table_Entry *****************************
 ******************************************************************************/
Cl_Lock_Table_Entry :: Cl_Lock_Table_Entry (int address)
{
  Address = address;
  Op_List_Granted = new T_Op_List ();
  Op_List_Waiting = new T_Op_List ();
}; // Cl_Lock_Table_Entry :: Cl_Lock_Table_Entry

// only empty entries will be deleted
Cl_Lock_Table_Entry :: ~Cl_Lock_Table_Entry ()
{
  delete Op_List_Granted;
  delete Op_List_Waiting;
}; // Cl_Lock_Table_Entry :: ~Cl_Lock_Table_Entry


/******************************************************************************
 ************ Function: Cl_Lock_Table_Entry :: request_Lock       ***************
 *******************************************************************************/
int
Cl_Lock_Table_Entry :: request_Lock (Cl_Op *op)
{
  Cl_Op  *op_tmp;
  int    list_length,i;
  int    reads=0;             // count read and write locks of other
  int    writes=0;            // transactions

  // no request on the data item: grant
  if (Op_List_Granted -> empty ())
    {
      if (! Op_List_Waiting->empty())
        {
          printf("Granted empty but not Waiting\n");
          //      report_statistics();
          exit(-1);
        };
#ifdef DEBUG
      cout << "grant write because no other lock " << op->get_Transaction_ID() << endl;
#endif
      op->set_Op_State(Lock_Grant);
      Op_List_Granted->append(op);
      return(LOCK_GRANTED);
    }
  else
    {
      // take a loot at the locks from other transactions already granted
      list_length = Op_List_Granted->length();
      for (i=0;i<list_length;i++)
        {
          op_tmp = Op_List_Granted->remove_front();
          if (op_tmp->get_Transaction_ID() != op->get_Transaction_ID())
```



```
            {
              switch (op_tmp->get_Op_Type())
                {
                case Read:
                  reads++;
                  break;
                case Write:
                  writes++;
                  break;
                }
              Op_List_Granted->append(op_tmp);
            };
        };
      // grant write if no other locks active
      if ((op->get_Op_Type() == Write) && ((writes+reads) == 0))
        {
#ifdef DEBUG
          cout << "write lock granted" << op->get_Transaction_ID() << endl;
#endif
          Op_List_Granted->append(op);
          op->set_Op_State(Lock_Grant);
          return(LOCK_GRANTED);
        };
      // grant read if no write lock active
      if ((op->get_Op_Type() == Read) && (writes == 0))
        {
#ifdef DEBUG
          cout << "read lock granted" << op->get_Transaction_ID() << endl;
#endif
          Op_List_Granted->append(op);
          op->set_Op_State(Lock_Grant);
          return(LOCK_GRANTED);
        }
      else
        {
          // has to wait until other locks ready
          Op_List_Waiting->append(op);
          op->set_Op_State(Lock_Wait);
#ifdef DEBUG
          cout << "lock must wait" << op->get_Transaction_ID() << endl;
#endif
          return(LOCK_NOT_GRANTED);
        };
    };
}; // Cl_Lock_Table_Entry :: request_Lock


/********************************************************************************
 ************ Function: Cl_Lock_Table_Entry :: release_Lock      ****************
 *******************************************************************************/
void
Cl_Lock_Table_Entry :: release_Lock(Cl_Op *op)
{
  int i;
  int list_length;
  Cl_Op *op_tmp, *op_tmp2;

#ifdef DEBUG
  cout << "entry release lock " << op->get_Transaction_ID() << endl;
#endif

  list_length = Op_List_Granted->length();
  for (i=0;i<list_length;i++)
    {
      op_tmp = Op_List_Granted->remove_front();
      if (op == op_tmp)
        break;
      else
```

```
        Op_List_Granted->append(op_tmp);
    }

  if (Op_List_Granted->empty())
    {
      // grant first waiting write or all reads
      if (!Op_List_Waiting -> empty())
        {
          op_tmp = Op_List_Waiting->remove_front();
          Op_List_Granted->append(op_tmp);
          op_tmp->set_Op_State(Lock_Grant);
#ifdef DEBUG
          cout << "grant first waiting lock " << op_tmp->get_Transaction_ID() << endl;
#endif
          if (op_tmp->get_Op_Type() == Read)
            {
              // is read -> grant all waiting reads
              list_length = Op_List_Waiting->length();
              for (i=0;i<list_length;i++)
                {
                  op_tmp = Op_List_Waiting->remove_front();
                  if (op_tmp->get_Op_Type() == Read)
                    {
#ifdef DEBUG
                      cout << "grant more reading locks " <<
                        op_tmp->get_Transaction_ID() << endl;
#endif
                      Op_List_Granted->append(op_tmp);
                      op_tmp->set_Op_State(Lock_Grant);
                    }
                  else
                    Op_List_Waiting->append(op_tmp);
                }; // for all waiting
            }; // is read
          Any_Lock_Granted->set();
        }; // someone waiting
    } // grant list empty
  else
    {
      // check whether only locks of one transaction are active and another lock of
      // this transaction is waiting
      op_tmp = Op_List_Granted->remove_front();
      list_length = Op_List_Granted->length();
      for (i=0;i<list_length;i++)
        {
          op_tmp2 = Op_List_Granted->remove_front();
          Op_List_Granted->append(op_tmp2);
          if (op_tmp->get_Transaction_ID() != op_tmp2->get_Transaction_ID())
            {
              Op_List_Granted->append(op_tmp);
              // locks of different transactions
              return;
            }
        }
      Op_List_Granted->append(op_tmp);
      // now look whether further locks of same transaction waiting
      if (! Op_List_Waiting->empty())
        {
          list_length = Op_List_Waiting->length();
          for (i=0;i<list_length;i++)
            {
              op_tmp2 = Op_List_Waiting->remove_front();
              if (op_tmp->get_Transaction_ID() == op_tmp2->get_Transaction_ID())
                {
                  // grant the lock
                  Op_List_Granted->append(op_tmp2);
                  op_tmp->set_Op_State(Lock_Grant);
                }
```



```
              else
                Op_List_Waiting->append(op_tmp2);
            }
          Any_Lock_Granted->set();
        }; // waiting list not empty
    }; // only one transaction has locks
}; // release lock

/*********************************************************************************
 *********** Function: Cl_Lock_Table_Entry :: remove_Wait       ****************
 *********************************************************************************
 Purpose:    removes a lock from waiting list
 ********************************************************************************/
void
Cl_Lock_Table_Entry :: remove_Wait(Cl_Op *op)
{
  int i;
  int list_length;
  Cl_Op *op_tmp;

  list_length = Op_List_Waiting->length();
  for (i=0;i<list_length;i++)
    {
      op_tmp = Op_List_Waiting->remove_front();
      if (op != op_tmp)
        Op_List_Waiting->append(op_tmp);
      else
        {
#ifdef DEBUG
          cout << "lock removed " << op_tmp->get_Transaction_ID() << endl;
#endif
          break;
        };
    };
};


/*********************************************************************************
 *************** Function: Cl_Lock_Table_Entry :: unused *********************
 *********************************************************************************
 Purpose:    Returns TRUE if both lists of this Lock_Table_Entry are empty, i.e.
 no operation is waiting for this lock any more.
 ********************************************************************************/
int
Cl_Lock_Table_Entry :: unused () {
  if (Op_List_Granted->empty () && Op_List_Waiting -> empty ())
    return(1);
  else
    return(0);
}; // Cl_Lock_Table_Entry :: unused

/*********************************************************************************
 *********************************************************************************
 ****************************** Cl_Lock_Table ********************************
 *********************************************************************************
 ********************************************************************************/
Cl_Lock_Table :: Cl_Lock_Table ()
{
  int i;                 // Index

  Hash_Table = new (T_Lock_List *) [LOCK_TABLE_SIZE];

  for (i = 0; i < LOCK_TABLE_SIZE; i++) {
    Hash_Table [i] = new T_Lock_List ();
  }; // for
}; // Cl_Lock_Table :: Cl_Lock_Table
```

```
// i will not implement it, because i do not plan to delete the lock table
Cl_Lock_Table :: ~Cl_Lock_Table ()
{}


/********************************************************************************
 ************ Function: Cl_Lock_Table_Entry :: find_Entry         ***************
 ********************************************************************************
 Purpose:    returns the entry of hash table for the data-item
 ********************************************************************************/
Cl_Lock_Table_Entry *
Cl_Lock_Table :: find_Entry(int address)
{
  int   hash_addr;
  int   number_entries;
  int   i;
  Cl_Lock_Table_Entry *entry;

  // Lock for the index into the lock hash table
  hash_addr = hash_Address (address);
  number_entries = Hash_Table [hash_addr] -> length ();

  for (i=0;i<number_entries;i++)
    {
      entry = Hash_Table [hash_addr] -> remove_front ();
      Hash_Table[hash_addr]->append(entry);
      if (entry -> get_Address () == address)
        {
          // Right lock found
          return entry;
        };
    };
  entry = new Cl_Lock_Table_Entry(address);
  Hash_Table[hash_addr]->append(entry);
  return entry;
}

/********************************************************************************
 ************ Function: Cl_Lock_Table_Entry :: remove_Entry         ***************
 ********************************************************************************/
void
Cl_Lock_Table :: remove_Entry(Cl_Lock_Table_Entry *entry)
{
  int   hash_addr;
  int   number_entries;
  int   i;
  Cl_Lock_Table_Entry *entry_tmp;

  // Lock for the index into the lock hash table
  hash_addr = hash_Address (entry->get_Address());
  number_entries = Hash_Table [hash_addr] -> length ();

  for (i=0;i<number_entries;i++)
    {
      entry_tmp = Hash_Table [hash_addr] -> remove_front ();
      if (entry == entry_tmp)
        {
          delete entry;
          return;
        }
      else
        Hash_Table[hash_addr]->append(entry_tmp);
    };
}; // Cl_Lock_Table : remove_Entry

/********************************************************************************
 ************* Function: Cl_Lock_Table :: request_Lock       ******************
 ********************************************************************************/
```



```
int
Cl_Lock_Table :: request_Lock (Cl_Op *op)
{
  int i;
  int address;
  int result;
  double timeAtBegin;

  Cl_Lock_Table_Entry *entry;

  //    cout << "l rrrr: Cl_Lock_Table :: request_Trans_Lock:" << endl;

  // Get the address of data item accessed
  address = op->get_Op_Address();
  entry = find_Entry(address);
  op->set_Lock_Entry_Ptr(entry);
  result = entry->request_Lock(op);
  if (result == LOCK_GRANTED)
    return(LOCK_GRANTED);
  else
    {
      timeAtBegin = simtime();
      while (1)
        {
          if (Any_Lock_Granted->timed_wait(timeAtBegin+DEADLOCK_TIME_OUT-simtime()) ==
TIMED_OUT)
            return(LOCK_NOT_GRANTED);
          else if (op->get_Op_State() == Lock_Grant)
            return(LOCK_GRANTED);
        };
    }; // wait for lock
}; // Cl_Lock_Table :: request_Write_Locks


/********************************************************************************
 ************* Function: Cl_Lock_Table :: release_Lock  ************************
 ********************************************************************************
 * Purpose: looks for the right entry and calls the release function of the entry
 ********************************************************************************/
void
Cl_Lock_Table :: release_Lock (Cl_Op *op)
{
  int address;
  Cl_Lock_Table_Entry *entry;

  entry = op->get_Lock_Entry_Ptr();
  entry->release_Lock(op);
  if (entry->unused())
    remove_Entry(entry);
}; // Cl_Lock_Table :: release_Locks

/********************************************************************************
 ************* Function: Cl_Lock_Table :: remove_Wait      **********************
 ********************************************************************************
 * Purpose: looks for the right entry and calls the remove function of the entry
 ********************************************************************************/
void
Cl_Lock_Table :: remove_Wait(Cl_Op *op)
{
  int address;
  Cl_Lock_Table_Entry *entry;

  // Get the address of data item accessed
  address = op->get_Op_Address();
  entry = find_Entry(address);
  entry->remove_Wait(op);
}; // Cl_Lock_Table :: remove_wait_Lock
```