Overview

A. Basic machinery

execution model, byte code vs. native code, language aspects

B. Memory management

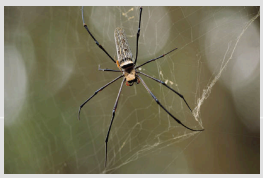
basic schemes, memory types: transient vs. persistent, garbage collection

C. Atomicity and transactions

basic schemes, system-level vs. user-level transactions

D. OO programming w/ resource constraints

applet design, RMI, size/performance optimizations



Copyright © 2004-2007 IBM Corp.

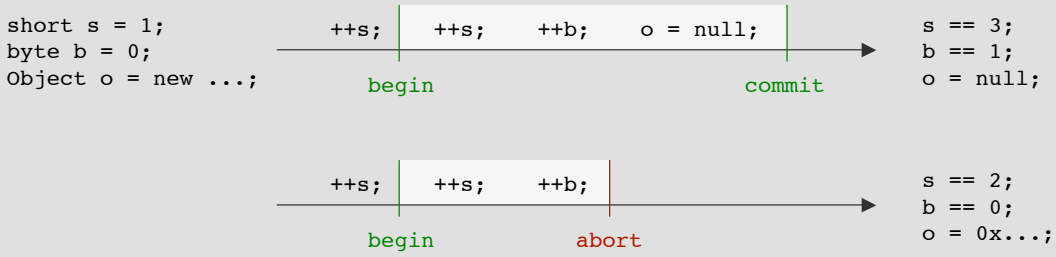
C. Transactions: Atomic Operations

- Definition
  - *an operation executes **atomically** if when it terminates normally all ist externally visible effects are made permanent, else it has no effect at all*
  - *in case of a failure during the execution of an atomic operation, the system can be rolled back to its prior state*
- Example: JavaCard
  - *Primary failure w/ smart cards*
    - premature removal of the card from the reader, so-called **tearing**
  - *Atomic operations in JavaCard*
    - updating persistent fields of primitive type (i.e., `boolean`, `byte`, `short`, `int`)
    - updating persistent object references
    - updating persistent array elements of primitve type or object references

Copyright © 2004-2007 IBM Corp.

C. Transactions: Composite Atomic Operations

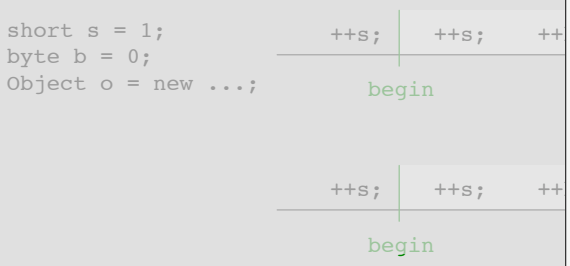
- Definition
  - *a **transaction** is an atomic update of several different fields in potentially different objects (composite atomic operation)*
  - *if the transaction terminates successfully, all ist externally visible effects are made permanent, else it has no effect at all*
  - *basic API:* `begin`, `commit`, `abort`
- Example



Copyright © 2004-2007 IBM Corp.

C. Transactions: Composite Atomic Operations

- Definition
  - *a **transaction** is an atomic update of several different fields in potentially different objects (composite atomic operation)*
  - *if the transaction terminates successfully, all ist externally visible effects are made permanent, else it has no effect at all*
  - *basic API:* `begin`, `commit`, `abort`
- Example



ACID

1. Atomicity
   either all of the tasks of a transaction are performed or none of them are
2. Consistency
   the system is in a legal state when the transaction begins and when it ends
3. Isolation
   operations within a transaction appear isolated from all other operations
4. Durability
   once successful, the transaction will persists and not be undone

Copyright © 2004-2007 IBM Corp.

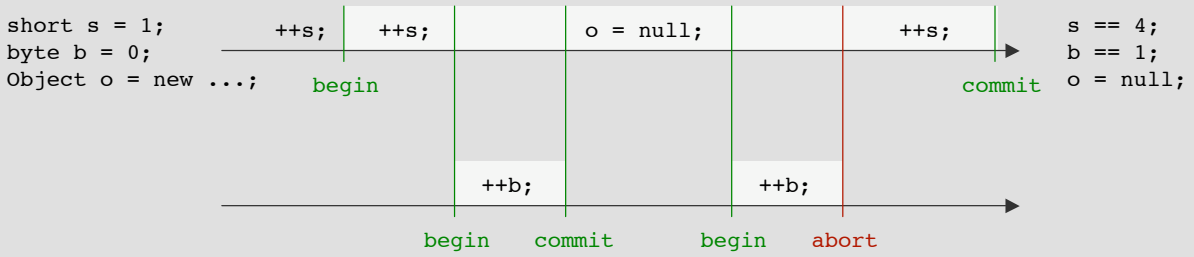
C. Transactions: Processing & Recovery Strategies

- Transaction processing strategies
  - *optimistic (write-thru)*
    - write new value to persistent memory, write 'initial' value to log
    - discard log on **`commit`**, restore values from log on **`abort`**
  - *pessimistic (write-back)*
    - write new value to log, leave 'initial' value untouched (read from log)
    - discard log on **`abort`**, write log to persistent memory on **`commit`**
- Transaction recovery strategies
  - *backward recovery: return to the last consistent state (cancel transaction)*
  - *forward recovery: redo successful transactions from the transaction log*

Copyright © 2004-2007 IBM Corp.

C. Transactions: Nested Transactions

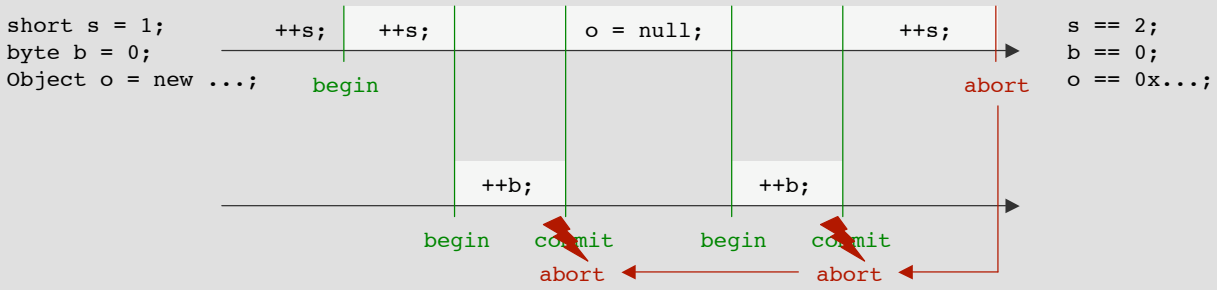
- Transactions within transactions
- Abort of an inner transaction does not abort its outer transaction
- Abort of an outer transactions aborts ist inner transactions



Copyright © 2004-2007 IBM Corp.

C. Transactions: Nested Transactions

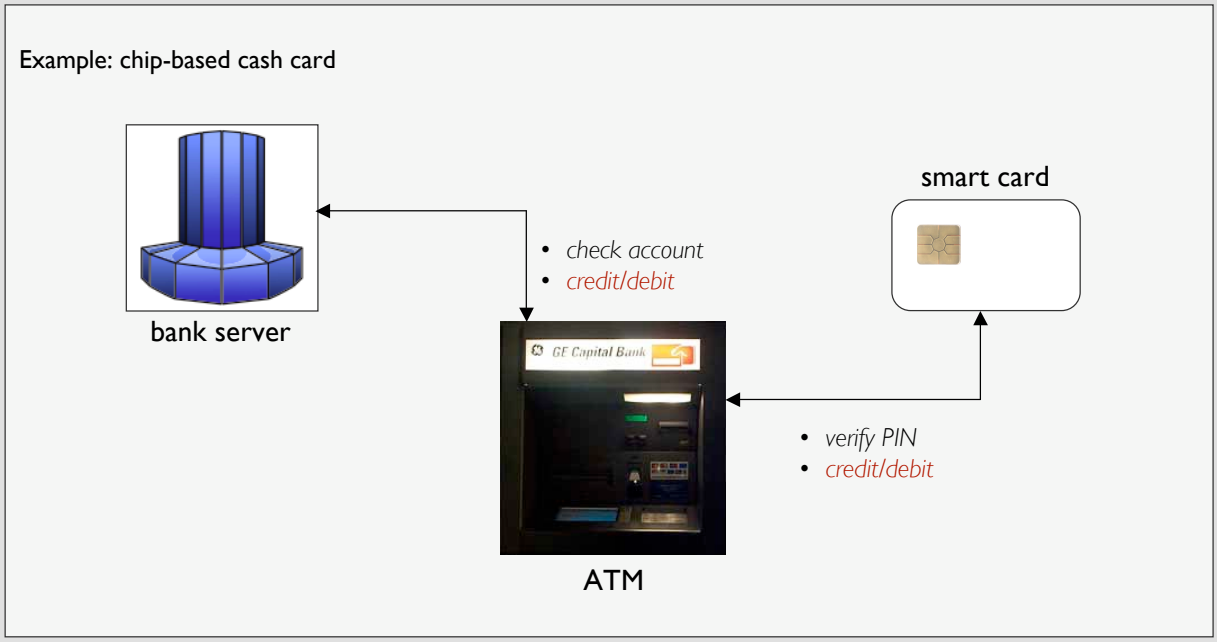
- Transactions within transactions
- Abort of an inner transaction does not abort its outer transaction
- Abort of an outer transactions aborts ist inner transactions



Copyright © 2004-2007 IBM Corp.

C. Transactions: Distributed Transactions

- Transaction spanning more than one (logical) process



Copyright © 2004-2007 IBM Corp.

C. Transactions: JavaCard

- EEPROM: optimistic w/ backward recovery
- ACD requires work, I is for free (no threads)
- Abort is default when
  - *voluntarily leaving* `Applet.process()` *while a transaction is running*
  - *uncaught exceptions cause* `Applet.process()` *to be left*
- Aborts do **not** restore transient fields or local variables
- System-level vs. user-level transactions
  - ***system level:*** *writing primitive data types, installing/deleting an applet, some API calls*
  - ***user level:*** *all transactions explicitly started by an applet*
- No nested user-level transactions

Copyright © 2004-2007 IBM Corp.

C. Transactions: JavaCard

- EEPROM: optimistic w/ backward recovery
- ACD requires work, I is for fre
- Abort is default when
  - *voluntarily leaving* `Applet.proc`
  - *uncaught exceptions cause* `Appl`
- Aborts do **not** restore transien
- System-level vs. user-level tran
  - ***system level:*** *writing primit some API ca*
  - ***user level:*** *all transactio*
- No nested user-level transacti

Optimistic w/ backward recovery

1. *in the wild, aborts happens rarely so optimize for successful transactions*
2. *reads happen far more often than writes (optimistic has faster reads)*
3. *with optimistic, the fields of objects allocated within a transaction a never logged*
4. *writing the log to persistent memory would be a transaction by itself*

Note: Flash basically requires some log-based memory architecture and, as such, transactions require much less additional effort.

Copyright © 2004-2007 IBM Corp.

C. Transactions: `javacard.framework`

- `javacard.framework.JCSystem`
  - *collection of methods to control applet execution, memory management, atomic transaction management, inter-applet object sharing [3.A]*
- `javacard.framework.Util`
  - *common static utility functions*

Copyright © 2004-2007 IBM Corp.

C. Transactions: `javacard.framework`

- `javacard.framework.JCSystem`
  - *collection of methods to control applet execution, memory management, atomic transaction manag*
- `javacard.framework.`
  - *common static utility funct*

```
public final class JCSystem {
    public static void beginTransaction();
    public static void commitTransaction();
    public static void abortTransaction();

    public static byte getTransactionDepth();

    public static short getMaxCommitCapacity();
    public static short getUnusedCommitCapacity();

    ...
};
```


Copyright © 2004-2007 IBM Corp.

C. Transactions: `javacard.framework`

- `javacard.framework.JCSystem`
  - *collection of methods to control applet execution, memory management, atomic transaction manag*
- `javacard.framework.`
  - *common static utility funct*

```
public class Util {
   public static short arrayCopy
         (byte[] src, short srcOfs,
          byte[] dest, short dstOfs, short length);
   public static short arrayCopyNonAtomic
         (byte[] src, short srcOfs,
          byte[] dest, short dstOfs, short length);

   public static short arrayFill(byte[] bArray,
         short offset, short length, byte value);
   public static short arrayFillNonAtomic
         (byte[] bArray, short offset,
         short length, byte value);

   ...
};
```


Copyright © 2004-2007 IBM Corp.

DEMO

Copyright © 2004-2007 IBM Corp.

C. Transactions: Sample

```
byte[] key_buffer = JCSystem.makeTransientByteArray(KEY_LENGTH,JCSystem.CLEAR_ON_RESET);
Object global_ref = new ClassA(); // persistent field

JCSystem.beginTransaction();
Util.arrayCopy(src,src_off,key_buffer,0,KEY_LENGTH);
Util.arrayCopyNonAtomic(src,src_off,key_buffer,0,KEY_LENGTH);
for (byte i = 0; i < KEY_LENGTH; ++i) key_buffer[i] = 0;
byte a_local = 1;
global_ref = new ClassB();
JCSystem.abortTransaction(); // only global_ref is restored

...

JCSystem.beginTransaction();
global_ref = JCSystem.makeTransientObjectArray(LENGTH,JCSystem.CLEAR_ON_DESELECT);
Object local_ref = new ClassC();
if (!condition) JCSystem.abortTransaction(); // global_ref is restored
else JCSystem.commitTransaction();
return local_ref; // potential dangling pointer, JCRE sets local_ref to null
```

[Chen. *Java Card Technology for Smart Cards.* Addison Wesley, 2000, pp. 62-63]

Copyright © 2004-2007 IBM Corp.