Lecture 24
Concurrent Programming

2nd December 2003

Remarks on programming projects

- Common error: almost everyone had this mistake which in my test case led to the receiver not getting all the messages the senders thought had been sent. Recall that every successful communication occurs when a thread doing a poll operation on an event matches with a waiting event. The data is passed from sender to receiver, the thread of the waiting event is notified to continue and the poll call returns. Problem: if the blocked event was part of a selection list there are blocked events on other channels associated with that selection list. If you don't do anything at this point, one of those other events may also participate in a successful communication! If that happens one or another of the two communications will be lost.

- Copying events: several of you copied the events from the selection list in order to put them on channels. I never considered this and still don't see why it is necessary. Discuss.

- Interesting semantics: several of you implemented semantics for select that removed successful events from selection lists. This wasn't what I had in mind but it seems like a reasonable alternative.

- Formatting: you all have CS degrees and should be able to format your code so that indentation is consistent and sensible. I am not picky about there being one "right way", but random indentation changes are not acceptable.

- Presentation and demonstration: when you complete a project you are responsible for convincing others that it works. Projects should always be accompanied by a description of what they do and how they do it as well as evidence that they work correctly for the intended application. If your code does not work when it is due, I expect to see an acknowledgement of that as well as your analysis of what you think might be wrong and things you have determined not to be wrong. For this project I suggested a single test that I considered a minimal demonstration of the program's correct behavior. If a teacher suggests a test case your

responsibility includes understanding what is the expected behavior of the test and recognizing when your program is not behaving as expected. Example: in a synchronous communication it is not reasonable to see hundreds of sends complete before the receive matching the first send. I was disappointed that no one undertook any creative steps to testing their code – and indeed some did even present evidence based on my suggested test. Brief executions perhaps under manual control are not even minimally adequate as evidence of correct function in concurrent programs where timing and synchronization errors are likely.

Remarks on the exam take-home

Much better, thank you. The barrier problem still gave people trouble. Discuss.

Joint Action Patterns

Suppose you want your bank to automatically move money from your savings to your checking account – whenever the checking account is below a certain threshold and the savings account is above a different threshhold.

Let's try to implement this as actions automatically taken either following a checking withdrawal or a savings deposit:

```
in the checking account object:
synchronized withdrawal(amt) {
   balance = balance - amt;
   synchronized (savings) {if balance<cmin_balance &&
      savings.balance>smin_balance {
         savings.withdraw(transferAmt);
         deposit(transferAmt);
   }}
}
synchronized deposit(amt) {...}
in the savings account object:
synchronized deposit(amt) {
   balance = balance + amt;
   synchronized (checking) {if balance>smin_balance &&
      checking.balance<cmin_balance {
         withdraw(transferAmt); (* how big should transferAmt be? *)
         checking.deposit(transferAmt);
   }}
}
synchronized withdraw(amt) {...}
```

What's wrong?

Techniques for fixing:

- let the race winner do the work – try to obtain a test-and-set lock – if you get the lock, check for and do the transfer, otherwise do nothing.
- decouple: let some other process monitor the balances and do the transfers
- decouple: fork a new thread to check the balances and do the transfer

Decoupling is a very useful strategy: its use may force one to rethink the operations. In our bank example, how do we really want the transfer to work? Process all the deposits, total the withdrawals (but don't do them), transfer funds, do the withdrawals.

Transactions

Transaction lifetime: begin, in-progress, {abort|commit}

Atomicity all or none

Consistency create a new valid state or restore state to what existed before it began; implicit here is the notion of a data invariant – a property that is true whenever we are not in the midst of processing a transaction.

Isolation effects of an uncommited transaction are not visible to other transactions

Durability results of committed transactions persist across system failure and restart

The transaction concept is most often associated with databases, but transactions are useful in other contexts as well, including contexts where one or more of the ACID properties can be relaxed.

Relation of transactions to processes or threads: transactions and threads may be in 1-1 correspondence, but need not be. In a distributed system, multiple threads per transaction are the norm. Some systems allow concurrency within a transaction, and others will multiplex several transactions onto a single thread.

Techniques for passing transaction information:

```
(* non - OO *)
tid = begin_transaction()
op1(tid, op1 parameters)
op2(tid, op2 parameters)
commit_transaction(tid)

(* OO *)
trans = new Transaction()
```

```
O1.op1(trans, op1 parameters)
...
trans.commit()

(* implicit *)
begin_transaction()
op1(op1 parameters)
...
commit_transaction()
```

General rules: each object (not thread) participating in a transaction has the right to abort the transaction at any time up until the time it promises to commit. Usually this policy is implemented by calling prepare-to-commit on each object (in optimistic systems) or by aborting at the first operation (in optimistic systems).

Source of optimistic and pessimistic terminology: optimistic in the sense that we assume that conflicts won't occur, run transaction to completion and then check for conflicts. Pessimistic, we assume that conflicts will occur and try to detect them early, stopping if one is detected. (Stopping may mean either aborting or waiting).

Pessimistic systems: (pessimistic because the assumption is that there will be conflicts and we want to avoid doing work that is not useful)

- abort (or wait) at first operation if a conflicting transaction is already present – either directly or on a referenced object; mark the object as belonging to this transaction
- at the first operation on an object, make a backup copy of the current state in case of abort
- in case of any failure, mark the current transaction as not committable
- can commit? – yes (unless transaction marked as not committable)
- to commit, mark the object not busy;
- to abort, restore previous state

Optimistic systems: (optimistic because the assumption is that conflicts are rare, so if we do extra work because of a conflicts it is not a big loss)

- at first operation on an object, create a copy of the state associated with the current transaction as ttransaction-local state
- when performing actions, perform them on the copied state
- in case of failure, mark the current transaction not comittable

- can commit? – only if no conflicting transactions have committed (or promised to commit); what is a conflicting transaction? One definition: a transaction whose write set intersects the touched (read+write) set of the current transaction.
- to commit, save transaction-local state as new state of object;
- to abort, throw away transaction-local state

Mixed optimistic and pessimistic strategies are also possible.

Non-database application

Dynamic code loading: an executing program loads new code into its address space. The code to be loaded may consist of several modules. The requirement is that when loading is complete all undefined symbols must be resolved, either to points in the existing code (which may have dynamically loaded components) or to points in the newly loaded set of modules.

A transactional strategy is appropriate here, even if there is no concurrency involved. Since the *load state* is potentially large, and we expect that programs will usually do the right thing (i.e. load sets of modules that meet the all-symbols-resolved test), we used an optimistic strategy for managing the state: the transaction treats the existing state as read-only and makes up new data structures representing the changes to the load state. If the loading is successful the new state is incorporated into the permanent state, otherwise it is thrown away.