

## Copyright warning

COMMONWEALTH OF AUSTRALIA  
Copyright Regulations 1969  
**WARNING**  
This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (the Act).  
The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.  
**Do not remove this notice.**

## COMP5348

### Lecture 2: State

Adapted with permission from presentations by Alan Fekete

## Outline

- Types of State
- Storing and accessing state
- Concurrency and its dangers
- Managing concurrent access to state

## State

- All imperative programming involves altering state (the values linked to names eg variables)
- In enterprise applications we distinguish several varieties of data that has state which is important

## Infrastructure state

- Some state is not connected with the business logic of the application, but instead provides general support for computation
  - Eg a pool of connections to the network
  - Eg a table of fonts

## Resource state

- Data about the real-world (domain about which the application operates)
  - Physical or organisational entities and their associations
    - Eg salary of an employee
    - Eg manager of a branch
  - Entities are instances within classes/types
  - Individual instances are identifiable

## Sessions and operations

- Session: A user does something they consider meaningful, over a period of time
  - Also called "business transaction"
  - Involving several back-and-forth interactions between user and application
  - Typical "use-case" for the application
  - Eg new employee starts work
- Operation: one step of the session, in which the application responds to one set of inputs from the user
  - Changes values of resource state, for one or more entities

## Session state

- To manage a session, application needs to keep state about it, from one operation to the next
  - Especially: where in the the session workflow
    - Eg: we have chosen the new employee id, captured their personal data, and created their account, but not yet set up their benefits package
  - Also: knowledge about the user (identity, role, privileges, etc)
  - Also: tentative changes to entities
    - Eg if this employee actually agrees to the package, we will reduce available funds in department budget

## Local computation state

- Temporary variables while performing an operation
  - Eg iterator into a collection we are scanning
  - Not usually described in requirements documents
    - But invented during coding

## Lifetime of state

- Resource state lasts from creation to explicit deletion, through many sessions
- Session state lasts from start of session to completion, through many operations
- Local computation state lasts during one operation

## Sharing of state

- Resource state means something about the domain
  - It will be accessed by many sessions
  - Possibly at the same time
- Session state is used by different steps of the session
  - Usually, one after another (but with gaps)
- Local computation state is not shared

## Caching

- Each piece of state should have one location where the authoritative information is always kept for this value
- However, copies are often kept elsewhere in the application
  - To be close to computation
- Eg when new employee is being created, we may fetch the department information, and use it repeatedly

## Cache management basics

- Fetch state on use, if not already present
  - Or prefetch based on assumed usage pattern
- Write-back on every modification
  - Or write-back later, if modified
- Invalidation of cached copies when authority is modified?

## Integrity of state

- There are invariants about state that need to be valid
  - If not, computation can misbehave
    - The invariant is a precondition for the code
  - Eg salary is always positive
  - Eg referential integrity: employee is in some department that exists
  - Eg sum of salaries of employees in department is less than total budget of department
- Note: invariants can be about one instance, or about several instances together

## Finding state

- Often, data about an entity is found based on the identifier of the instance
  - Eg promote\_staff takes staffid as argument
  - This can be very efficient eg hash-map
- Data about associated entities is found from cross-references in the original entity
- Sometimes, property-based access can be offered
  - Eg find employees whose address is in Bondi
  - This can be inefficient unless an index is kept

## State management techniques

- Where is the state kept?
  - Application code runs for one operation, and then stops until another request arrives
  - Resource state clearly needs to be kept externally, so it can be shared among different users
  - For session state, there is the difficulty of making sure that subsequent requests carry on from what was left after the previous request of this particular session
    - Correlation problem
  - The application code may be invoked from many different sessions
    - It is wise to also keep session state in an external persistent form

## Persistent state

- Persistent state that is outside the application could be in a file (managed by OS) or in a database (managed by dbms) or in persistent messages (managed by MOM) or in the client
- The application code needs to get state in and out
  - It keeps program objects that are cached copies of the state
    - Inter-object references are not the same for in-program than for externally stored formats
  - Application needs special code to parse and unparse the external formats
    - Eg Java serialize

## State from a database

- Especially for resource data
- Application usually has a data access layer
  - Connects to the database
  - Executes SQL to retrieve the data
  - Builds program objects (eg "J2EE entity beans")
  - Takes care of writing back modifications
- Many frameworks to generate this code automatically (from just eg the SQL table definitions)
  - Object class corresponds exactly to one row of a table
  - Or object class for a collection of rows
  - Or object class for a row and associated information (join)

## Session state techniques

- Session state may be stored in a database too
- Special tables for eg workflow status
- Tentative changes to entities
  - Either special status fields in the same table as resource data
  - Or, more usually, separate table(s) for tentative information
    - With similar structure to committed data table

## Passing session state around

- Instead of storing session state in a dbms, the state may be kept by the client, and supplied in each operation request
  - “cookies” do this automatically with web clients, but depend on browser settings
  - Instead, session state can be explicitly passed back and forth in each request and response, as extra arguments
  - Or session state can be encoded in the URL (RESTful system design, see later)

## Example

- Student records system

## Concurrency

- Many situations where several computations that share data can run “at the same time”
  - True parallel execution (eg on multiple nodes or on several cores in one machine)
  - Interleaving at different granularities of view (eg computations swap in and out)

## Why concurrency?

- Users may be simultaneously active
  - Two students using the student record system at lunchtime
- Concurrency may be introduced to improve performance
  - Do something useful in one computation while another is waiting

## Mechanisms for concurrency

- Multiple machines
- Separate processes in one machine
- Threads (supported by OS) within a process
- Threads (supported by runtime language environment)
- The issues we discuss apply in all of these

## Interleaving

- Each computation follows the code that is written for it
  - Line by line, one after another
- Observed from outside, the computations are interleaved
  - Computation 1 does steps C1.1 then C1.2 then C1.3
  - Now Computation 2 does C2.1 then C2.2
  - Computation 1 resumes with C1.4 then C1.5
  - Computation 2 resumes with C2.3

## Concurrency problems

- When interleaved computations have any shared state that they operate on, things can go badly wrong
- The state on which the computation depends can be changed unexpectedly
  - By another thread, between steps of the computation

## Lost update

- T1 wants to increase x by 1, and T2 wants to increase x by 2
- T1 sees x=10
  - T2 sees x=10
  - T2 sets x=12
- T1 sets x=11
- Final state is increased by 1 not by 3

## Check-use gap

- T1 checks that the room has space (`class.enrollment < class.room.capacity`)
- T2 sees that the class is underfull and reallocates it to another room (`class.room = new_room`)
- T1 inserts a new student (`class.enrollment = class.enrollment+1`)

## Inconsistent read

- T1 wants to calculate total load for INFO units; T2 changes student 53 from INFO1103 to INFO1003
- T1 reads `info1003.enrollment` (100)
- T2 does `info1003.enrollment++` (now 101)
- T2 does `info1103.enrollment--` (now 89)
- T1 reads `info1103.enrollment` (89)
- T1 returns 189 (not correct answer of 190)

## Immutable data

- Concurrency problems can be avoided if data never changes once it is initialized
- “append-only” data
- This leads to a different coding style, and it can be very inefficient in memory (continually creating new objects)

## “Stateless” code

- Concurrency problems can be avoided if code doesn't use shared data
- Purely functional computation

## Concurrency control

- For realistic applications, concurrency and shared data are both unavoidable
- So need mechanisms to restrict the interleavings and prevent bad things happening
  - Without reducing to a single-threaded computation
  - So allow harmless concurrency, but not harmful concurrency

## Pessimistic concurrency control

- Key idea: before touching a shared item, thread must obtain ownership of a “lock” that records that the shared item is being touched
- The lock comes with built-in mechanism that stops a second thread gaining ownership while another already owns it
- A second thread that tries to get ownership is blocked, takes no steps until the lock is available

## Location of concurrency control

- DBMS has locking built-in
- Multicore hardware has some instructions which do locking built-in
- Applications can write locking, using OS or RTE primitives

## Locks and blocking

- The impact of locking is to sometimes block a computation (temporarily)
- While the computation is blocked, other threads can make progress and use system resources
- There should be enough threads in the system to avoid wasted time
- But not too many threads, or managing them takes too much overhead

## Deadlock

- If a thread is blocked while it holds other locks (or other system resources), then it may itself be delaying other threads for even longer
- If there is a cycle of waiting, none of them will ever make progress
  - System “hangs”
- Deadlock may involve locks that are in different system layers

## Granularity of locks

- Decision of what lock to use for a shared piece of state requires a convention
- DBMS has built-in lock for each record in a table
- Changing the convention may improve performance a lot
  - Eg if several pieces of state are protected by same lock, less overhead in obtaining locks and less chance of deadlock
  - But sometimes harmless concurrency is prevented, leading to wasted time
  - Lots of issues for tuning design for best performance

## Optimistic concurrency control

- Instead of blocking when access may be dangerous, allow access to proceed
  - Then check afterwards, if any harm occurred
  - If so, remove computation
- This may waste work of unsuccessful computation
- But at low contention it often performs well
- This is especially easy to program in the application
  - Keep a modification count in the object
  - Look at it when you read the object, remember it
  - Check it hasn't changed when you later try to change the object

## Summary

- Resource state, session state, local computation state
  - Persistent state can be stored in databases, or passed around the system
  - State is often cached

## Summary

- Concurrency is a reality
  - Interleaved access to shared state is dangerous
  - Know the techniques that can avoid the problems
    - Sometimes you can use what the infrastructure provides
    - Often you have to code your own in the application
    - But they all have risks of their own

## Further reading

- M. Fowler "Patterns of Enterprise Application Architecture" (Addison-Wesley 2003)
- B. Goetz "Java Concurrency in Practice" (Addison-Wesley 2006)
- P. Helland "Data on the Outside vs Data on the Inside" Proc CIDR 2005
  - Much more on P. Helland's blog <http://blogs.msdn.com/pathelland/default.aspx>